

# В чем основная идея?

ASP.NET MVC является фреймворком для разработки от Microsoft, который сочетает в себе эффективность и аккуратность архитектуры MVC, самые современные идеи и методы гибкой разработки и лучшие свойства существующей платформы ASP.NET. Это альтернатива традиционным ASP.NET Web Forms, которая обеспечивает существенное преимущество для всех, кроме самых простых и тривиальных, проектов веб-разработки. В этой главе вы узнаете, почему Microsoft изначально создал ASP.NET MVC, что он представляет собой по сравнению со своими предшественниками и альтернативами, и, наконец, что нового появилось в ASP.NET MVC 4.

## Краткая история веб разработки

Для того чтобы понять различные аспекты и дизайнерские задачи ASP.NET MVC, стоит рассмотреть историю веб разработки хотя бы вкратце. На протяжении многих лет платформы для веб разработки от Microsoft демонстрировали возрастающую мощность и, к сожалению, возрастающую сложность. Как показано в [таблице 1-1](#), каждая новая платформа устранила конкретные недостатки своего предшественника.

**Таблица 1-1:** Родословная технологий веб разработки от Microsoft

Период	Технология	Сильные стороны	Слабые стороны
Юрский период	Common Gateway Interface («общий интерфейс шлюза»), CGI	Простота, гибкость и единственный вариант на то время	Работает вне веб сервера, таким образом, является ресурсоемким (использует отдельный процесс операционной системы для каждого запроса)
Бронзовый век	Microsoft Internet Database Connector, IDC (коннектор баз данных для Интернета)	Работает внутри веб сервера	Просто оболочка для SQL запросов и шаблонов для форматирования множества результатов
1996	Active Server Pages, ASP (активные серверные страницы)	Общая цель	Интерпретируется во время выполнения; поддерживает "спагетти-код"
2002/03	ASP.NET Web Forms 1.0/1.1	Скомпилированный; UI, сохраняющий состояние (stateful); обширная инфраструктура; поддерживает объектно-ориентированное программирование	Тяжелый по пропускной способности; некрасивый HTML; нетестируемый
2005	ASP.NET Web Forms 2.0		
2007	ASP.NET AJAX		
2008	ASP.NET Web Forms 3.5		

Период	Технология	Сильные стороны	Слабые стороны
2009	ASP.NET MVC 1.0		
	ASP.NET MVC		
2010	2.0, ASP.NET Web Forms 4.0		
2011	ASP.NET MVC 3.0		
	ASP.NET MVC		
2012	4.0, ASP.NET Web Forms 4.5		

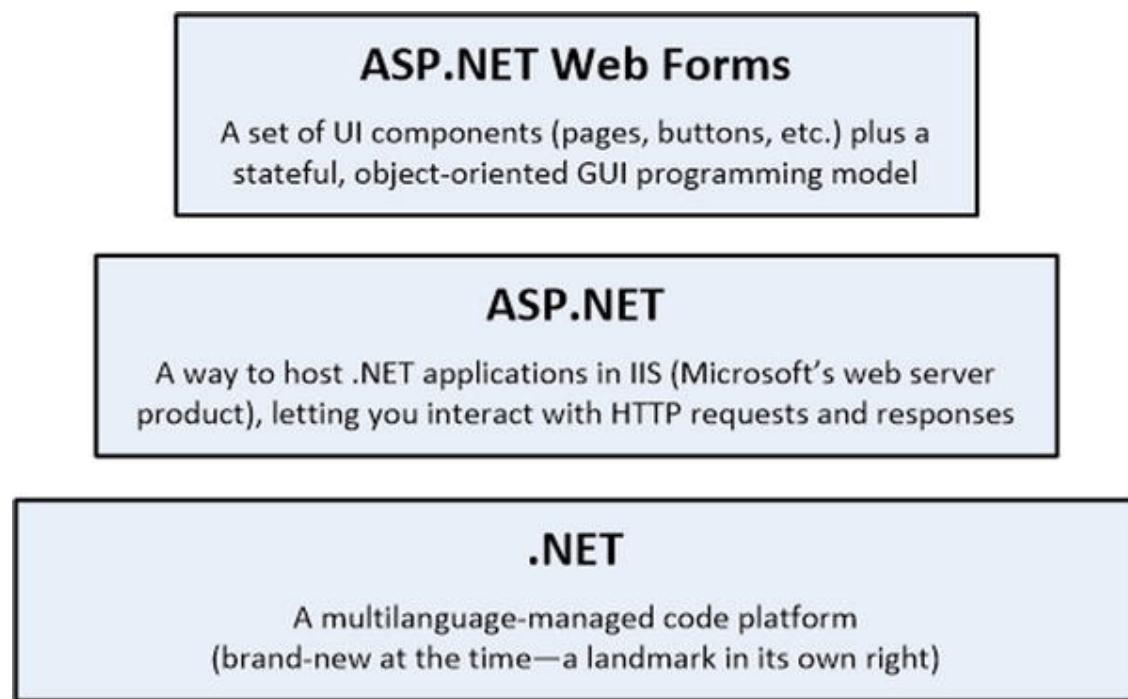
### Примечание

CGI является стандартным средством подключения веб сервера к произвольно выполняемой программе, которая возвращает динамический контент. Спецификация поддерживается Национальным центром суперкомпьютерных приложений (NCSA).

### Традиционные ASP.NET веб формы

ASP.NET являлся огромным прорывом, когда впервые появился в 2002 году. На [рисунке 1-1](#) показан стек технологий Microsoft, как мы теперь их наблюдаем.

**Рисунок 1-1:** Стек технологий ASP.NET Web Forms



В Web Froms Microsoft попытался спрятать работу как с HTTP, так и с HTML (которые в то время были незнакомы для многих разработчиков) с помощью моделирования пользовательского интерфейса (UI) в виде иерархии объектов, управляемых со стороны сервера. Каждый элемент управления следил за своим состоянием при всех запросах (с помощью возможности View State), если нужно, обрабатывая себя как HTML и автоматически подключаясь к событиям на стороне клиента (например, к нажатию кнопки) с соответствующим кодом для обработчика событий на серверной стороне. По сути, веб формы представляют собой гигантский слой абстракции,

предназначенный для доставки классического событийного графического интерфейса пользователя (GUI) через Интернет.

Идея заключалась в том, чтобы сделать веб разработку наподобие разработки Windows Forms. Разработчикам больше не нужно было бы работать с серией независимых HTTP запросов и ответов; мы смогли бы работать в условиях UI, сохраняющих состояние (*stateful*). Мы могли бы забыть о Сети и ее природе "несохранения состояния" и вместо этого выстраивать пользовательские интерфейсы при помощи конструктора «drag-and-drop», и представьте себе, или по крайней мере сделайте вид, что все это происходит на сервере.

## Что не так с ASP.NET Web Forms?

В принципе, традиционная разработка ASP.NET Web Forms была очень классной, но реальность оказалась более требовательной. Со временем использование веб форм в реальных проектах показало некоторые их недостатки:

- *Бес View State*: В результате использования актуального механизма для поддержки состояния между запросами (известного как View State) мы получили большие блоки данных, передаваемые между клиентом и сервером. Эти данные могут достигать сотен килобайт даже для скромных веб приложений, и они идут туда и обратно при *каждом* запросе, что приводит к увеличению времени отклика и повышению требований к пропускной способности сервера.
- *Жизненный цикл страницы*: Механизм для объединения события со стороны клиента с кодом серверного обработчика события – часть жизненного цикла страницы – может быть чрезвычайно сложным и деликатным. Немногие разработчики добились успеха в манипуляциях с элементами управления во время выполнения кода, не получив ошибок View State или не обнаружив, что некоторые обработчики событий таинственным образом не выполнялись.
- *Неправильное разделение задач*: Модель выделенного кода (code-behind) ASP.NET предоставляет возможность для того, чтобы вынести код приложения за рамки HTML разметки в отдельный класс выделенного кода. Это широко приветствовалось из-за разделения логики и представления, но, в действительности, разработчики вынуждены смешивать код представления (например, манипуляции с деревом серверных элементов управления) с логикой приложения (например, управлением базами данных) в этих же классах выделенного кода, которые становятся просто чудовищными. Конечный результат может быть недолговечным и непонятным.
- *Ограниченные возможности с HTML*: Серверные элементы управления отображают себя как HTML, но не обязательно так, как вы хотите. До версии ASP.NET 4 выходным данным HTML не удавалось соответствовать веб стандартам или хорошо работать с каскадными таблицами стилей (CSS). Также серверные элементы управления генерировали непредсказуемые и сложные значения атрибута ID, к которым трудно получить доступ при помощи JavaScript. Эти проблемы во многом решились в ASP.NET 4 и ASP.NET 4.5, но у вас все еще могут возникнуть сложности в получении того HTML, который вы ожидаете.
- *Абстракции с брешью*: Web Forms пытается спрятать HTML и HTTP, где это только возможно. Когда вы пытаетесь реализовать пользовательские механизмы поведения, вы часто можете выпасть из абстракций, которые заставляют вас переделывать механизм обратной передачи событий или выполнять глупые действия, чтобы он сгенерировал желаемый HTML. Кроме того, все эти абстракции могут стать неприятным барьером для компетентных веб разработчиков.
- *Слабая тестируемость*: Разработчики ASP.NET не могли предположить, что автоматизированное тестирование станет важным компонентом разработки программного обеспечения. Не удивительно, что жесткая архитектура, которую они разработали, не

подходит для модульного тестирования (юнит-тестирования). С интеграционным тестированием также могут возникнуть проблемы.

ASP.NET продолжал развиваться. В версии 2.0 был добавлен набор стандартных компонентов для приложений, и они могут уменьшить объем кода, который вам нужно писать самостоятельно. Релиз AJAX в 2007 году был ответом Microsoft на безумие Web 2.0/AJAX, он поддерживал хорошее взаимодействие со стороной клиента, не усложняя жизнь разработчикам. Все намного улучшилось с релизом ASP.NET 4, который впервые самым серьезным образом принял веб стандарт. Самый последний релиз, ASP.NET 4.5, на самом деле, имеет некоторые черты ASP.NET MVC и применяет их к миру Web Forms, что помогает решить некоторые довольно значимые проблемы, но, несмотря на это, многие внутренние ограничения все же присутствуют.

## Веб разработка сегодня

Вне Microsoft технологии веб разработки довольно быстро прогрессируют в нескольких различных направлениях, с тех пор как была впервые выпущена Web Forms. Кроме AJAX появились и другие важные технологии.

### Веб стандарты и REST

В последние годы возросла потребность соответствовать веб стандартам. Веб сайты предназначаются для самых разных устройств и браузеров, чем когда-либо прежде, и веб стандарты (HTML, CSS, JavaScript и т. д.) остаются нашей одной большой надеждой наслаждаться приличным просмотром сайтов и приложений везде, даже по холодильникам с интернет поддержкой. Современные Web платформы не могут позволить себе игнорировать идеи бизнеса и желания разработчиков соответствовать веб стандартам.

Повсеместно начинает использоваться HTML5. Он предоставляет веб разработчикам богатые возможности, что позволяют выполнять работу, которая ранее была исключительной прерогативой серверов, на стороне клиента. Эти новые возможности и наступающая зрелость библиотек JavaScript, таких как JQuery, JQuery UI, и JQuery Mobile, обозначают, что стандарты становятся все более важными и составляют основу для более богатых веб приложений.

### Примечание

В этой книге мы коснемся HTML5, JQuery и схожих вещей, но мы не будем вдаваться в подробности, потому что эти технологии сами по себе заслуживают особого внимания. Если вы хотите получить более полную информацию, то вы можете найти книги Адама Фримана, изданные Apress, по этим темам: *Pro jQuery*, *Pro JavaScript for Web Apps* (Pro JavaScript для веб приложений) и *The Definitive Guide to HTML5* (Полное руководство по HTML5).

В то же время REST (Representational State Transfer, «передача представлений состояний») стал доминирующей архитектурой для взаимодействия приложений через HTTP, полностью затмевая SOAP (технологию, лежащую в основе оригинального подхода ASP.NET к веб сервисам). REST описывает приложение в условиях ресурсов (URI), представляя реальные объекты и стандартные операции (HTTP методы), отражая доступные операции на этих ресурсах. Например, вы можете использовать PUT с новым `http://www.example.com/Products/Lawnmower` или DELETE с `http://www.example.com/Customers/Arnold-Smith`

Современные веб приложения обслуживают не только HTML. Зачастую они также должны обслуживать данные JSON или XML для различных технологий клиента, включая AJAX,

Silverlight, и родных приложений для смартфонов. Это происходит, естественно, с REST, который устраняет исторические различия между веб сервисами и веб приложениями, но требует такого подхода к обработке HTTP и URL, который не так легко поддерживается ASP.NET Web Forms.

## Экстремальное программирование и разработка через тестирование (test-driven development, TDD)

Не только веб разработка развивалась в последнее десятилетие – разработка программного обеспечения, в целом, также смешилась на путь *экстремального* программирования. Это может означать много разных вещей, но в основном речь идет о создании программных проектов как адаптационных процессов открытия и разработки без обременительного и ограничивающего перспективы планирования. Желание экстремального программирования, как правило, идет рука об руку с определенным набором программистских навыков и используемых средств (как правило, с открытым исходным кодом), которые содействуют и помогают изучению и использованию таких технологий.

*TDD* (разработка через тестирование) и его последнее воплощение, *BDD*, являются двумя очевидными примерами. Идея заключается в разработке программного обеспечения с изначального описания примеров желаемого поведения (известного как *тесты* или *спецификации*), так что в любой момент вы можете проверить стабильность и правильность вашего приложения, выполняя набор тестов по отношению к реализации. .NET инструментов хватает для поддержки TDD/BDD, но они, как правило, не очень хорошо работают с Web Forms:

- *Инструменты для модульного тестирования* позволяют определить поведение отдельных классов или других небольших частей кода в изоляции. Они могут быть эффективно использованы только для программного обеспечения, которое было спроектировано как набор независимых модулей, так что каждый тест может быть запущен отдельно. К сожалению, немногие приложения Web Forms могут быть протестированы таким образом. Следуя руководству фреймворка о том, чтобы вставить логику в обработчики событий или даже использовать серверные элементы управления, которые непосредственно отправляют запросы к базам данных, разработчики обычно, в конечном итоге, плотно связывают логику приложения со средой выполнения Web Forms. Это смерть для модульного тестирования.
- *Инструменты автоматизированного тестирования пользовательского интерфейса* позволяют моделировать ряд взаимодействий пользователя с запущенным экземпляром приложения. В теории эти тесты могут быть использованы с Web Forms, но они могут не сработать, когда вы сделаете небольшие изменения в макете страницы. Не обращая на это внимание, Web Forms начинает генерировать совершенно другие HTML структуры и ID элементов, так что существующий набор тестов становится бесполезным.

Сообщество создателей .NET приложений с открытым исходным кодом и независимый поставщик программного обеспечения (ISV) создали высококачественные фреймворки для модульного тестирования ( NUnit и XUnit ), фиктивные фреймворки ( Moq и Rhino Mocks ), инверсионные контейнеры ( Ninject и Autofac ), серверы непрерывной интеграции ( Cruise Control и TeamCity ), объектно-реляционные мапперы ( NHibernate и Subsonic ) и тому подобное. Сторонники этих инструментов и методов подали свой голос, создавая публикации и организуя конференции под общим брендом ALT.NET . Традиционная технология ASP.NET Web Forms не поддерживает эти инструменты и методы из-за своего монолитного дизайна, поэтому у этой группы экспертов Web Forms не пользуется особым уважением.

## Ruby on Rails

В 2004 году Ruby on Rails был не особенно известной технологией с открытым исходным кодом от неизвестного игрока. И вдруг пришла слава, которая перевернула правила веб-разработки. Не то, что бы Ruby на Rails содержал революционные технологии, но эта концепция взяла существующие ингредиенты и смешала их таким убедительным и привлекательным образом, что существующие платформы начали гореть для стыда.

Ruby on Rails (или просто Rails, как его обычно называют) принял архитектуру MVC (которую мы опишем чуть ниже). Применяя MVC и работая в гармонии с HTTP-протоколом, а не против него, содействуя конвенциям вместо потребности в конфигурации и интегрируя инструментарий объектно-реляционного маппинга (ORM) в своей основе, приложения Rails заняли свое место без особых усилий. Это выглядело так, как должна была бы выглядеть веб разработка все время, как если бы мы вдруг поняли, что наши инструменты все эти годы воевали, а теперь война закончилась.

Rails показывает, что соответствие веб стандартам и REST не должно быть жестким. Он также показывает, что экстремальное программирование и TDD работают лучше всего с фреймворками, которые их поддерживают. Остальной мир веб разработки до сих пор к этому подтягивается.

## Sinatra

Благодаря Rails вскоре многие веб разработчики стали использовать Ruby в качестве основного языка программирования. Но в таком интенсивно развивающемся сообществе это было только вопросом времени, когда появится альтернатива Rails. Самая известная, Sinatra, появилась в 2007 году.

Sinatra отбрасывает почти всю стандартную инфраструктуру Rails (маршрутизацию, контроллеры, представления и т.д.) и просто картирует URL шаблоны для блоков кода Ruby. Посетитель запрашивает URL, вызывающий блок кода Ruby, который будет выполнен, и данные передаются обратно браузеру – вот и все. Это невероятно простой вид веб разработки, но он нашел свою нишу в двух основных направлениях. Во-первых, для поддерживающих REST веб сервисов он просто быстро выполняет свою работу (мы коснемся REST в главе 25). Во-вторых, поскольку Sinatra может быть подключена к широкому спектру HTML шаблонов с открытым исходным кодом и ORM технологиям, она часто используется в качестве основы, на которой собирается пользовательский Web фреймворк в соответствии с архитектурными потребностями любого проекта, который есть под рукой.

Sinatra еще предстоит отвоевать свою долю рынка у серьезных MVC платформ, таких как Rails (или ASP.NET MVC). Мы упоминаем ее здесь лишь для иллюстрации текущих тенденций индустрии веб разработки в сторону упрощения, а также потому что Sinatra выступает в качестве противоположной силы относительно других фреймворков, все более накапливая в себе основной функционал.

## Node.js

Другой важной тенденцией является движение в сторону использования JavaScript в качестве основного языка программирования. AJAX впервые показал нам, что JavaScript очень важен, JQuery показал нам, что он может быть мощным и элегантным, а движок Google JavaScript V8 с открытым исходным кодом показал нам, что он может быть невероятно быстрым. Сегодня JavaScript становится серьезным серверным языком программирования. Он служит хранилищем данных и является языком запросов для нескольких нереляционных баз данных, в том числе CouchDB и Mongo. Также он используется в качестве универсального языка на серверных платформах, таких как Node.js.

Node.js появился примерно в 2009 году и получил широкое признание очень быстро. Архитектурно он похоже на Sinatra в том, что он не применяет MVC паттерн. Это более низкоуровневый способ подключения HTTP запросов к коду. Его основные нововведения заключаются в следующем:

- *Использование JavaScript*: Разработчикам нужно работать только с одним языком, от клиентского кода до логики на стороне сервера и даже до логики запросов данных с помощью CouchDB и тому подобного.
- *Абсолютная асинхронность*: Базовый API Node.js просто не блокирует потоки во время ожидания ввода/вывода (I/O) или во время любой другой операции. Все I/O осуществляются с началом операции и затем получением обратного вызова, когда I/O завершается. Это означает, что Node.js делает чрезвычайно эффективным использование системных ресурсов и может обрабатывать десятки тысяч одновременных запросов для CPU (альтернативные платформы, как правило, ограничиваются около сотней одновременных запросов для CPU).

Как и Sinatra, Node.js является нишевой технологией. Обычно большинство бизнес проектов, создавая реальные приложения в ограниченных временных рамках, нуждаются в полных фреймворках, таких как Ruby on Rails и ASP.NET MVC. Node.js упоминается здесь только для того, чтобы мы могли рассмотреть другие современные технологии, а не только ASP.NET MVC. Например, ASP.NET MVC включает в себя асинхронные контроллеры (которые мы опишем в главе 17). Это способ обработки HTTP-запросов с неблокируемым I/O и обработки большего числа запросов для CPU. Вы увидите, что ASP.NET MVC очень хорошо интегрируется со сложным JavaScript кодом, работающим в браузере.

# Основные преимущества ASP.NET MVC

ASP.NET стал большим коммерческим успехом, но, как уже говорилось, остальной мир веб-разработки также развивался, и хотя Microsoft сдувал пыль с Web Forms, ее основные конструкции начали выглядеть весьма устаревшими.

В октябре 2007 года на первой конференции по ALT.NET в Остине, штат Техас, вице-президент Microsoft Скотт Гатри объявил и продемонстрировал новую платформу по разработке MVC, построенную на базовой платформе ASP.NET, явно задуманную как прямой ответ на эволюцию технологий, таких как Rails и как реакцию на критику Web Forms. В следующих разделах описывается, как эта новая платформа преодолела ограничения Web Forms и снова возвеличила ASP.NET.

## Архитектура MVC

Важно различать архитектурный паттерн MVC и ASP.NET MVC Framework. MVC паттерн не является новым, его корни уходят к 1978 году и проекту Smalltalk в Xerox PARC, но он завоевала огромную популярность сегодня в качестве паттерна для веб приложений по следующим причинам:

- Взаимодействие пользователя с MVC приложением следует естественному циклу: пользователь совершает действие, в ответ на это приложение меняет свою модель данных и предоставляет пользователю обновленный вид. А затем цикл повторяется. Это очень удобно для веб-приложений, предоставляемых в виде серии HTTP запросов и ответов.
- Необходимость веб приложению объединять несколько технологий (например, базы данных, HTML и исполняемый код), как правило, разбивается на множество уровней или слоев. Моделей, которые вытекают из этих комбинаций, естественны для концепции MVC.

ASP.NET MVC Framework реализует MVC паттерн и, тем самым, обеспечивает значительно улучшенное разделение концепций. На самом деле ASP.NET MVC реализует современный вариант MVC паттерна, который особенно хорошо подходит для веб приложений. Вы узнаете больше о теории и практике применения этой архитектуры в главе 3.

Применяя и адаптируя MVC паттерн, ASP.NET MVC Framework сильно конкурирует с Ruby on Rails и аналогичными платформами, и переносит MVC паттерн в основное русло мира .NET. Суммируя опыт и лучшую практику разработчиков, использующих другие платформы, можно сказать, что ASP.NET MVC может предложить даже больше, чем Rails.

## Расширяемость

Внутренние компоненты настольного ПК являются независимыми частями, которые взаимодействуют только через стандартные, публично документированные интерфейсы. Вы можете легко вынуть видеокарту или жесткий диск и заменить его другим от другого производителя и будете уверены, что он впишется в слот и будет работать. MVC Framework также построен как ряд независимых компонентов, удовлетворяющих .NET интерфейс или построенных на абстрактном базовом классе, так что вы можете легко заменить компоненты, такие как система маршрутизации, движок для просмотра и так далее другими.

ASP.NET MVC дизайнеры построили его таким образом, чтобы дать вам три варианта выбора для каждого компонента MVC Framework:

- Использовать реализацию *по умолчанию* компонента в его нынешнем виде (чего должно быть достаточно для большинства приложений).
- Вывести *подкласс* реализации по умолчанию для настройки ее поведения.
- Заменить компонент полностью при помощи новой реализации интерфейса или абстрактного базового класса.

Это похоже на модель еще из ASP.NET 2.0, но давайте пойдем еще дальше – прямо в сердце MVC Framework. Вы узнаете все о различных компонентах, как и почему вы, возможно, захотите настроить или заменить каждый из них, начиная с главы 12.

## Жесткий контроль над HTML и HTTP

ASP.NET MVC признает важность получения чистой, соответствующей стандартам разметки. Его встроенные методы HTML помощника предоставляют соответствующие стандартам выходные данные, но есть и более значительные философские изменений по сравнению с Web Forms. Вместо того чтобы плодить огромные участки HTML, котором нам сложно управлять, MVC Framework рекомендует вам выработать простой, элегантный стиль разметки с помощью CSS.

Конечно, если вы хотите использовать некоторые готовые виджеты для сложных элементов пользовательского интерфейса, такие как выбор даты или каскадное меню, то вам стоит знать, что подход ASP.NET MVC к разметке упрощает использование лучших в своем роде UI библиотек, таких как JQuery UI или библиотеки Yahoo YUI. Разработчики JavaScript будут рады узнать, что ASP.NET MVC так хорошо сработался с популярной библиотеки JQuery, что Microsoft сделал JQuery встроенной частью шаблона проектов ASP.NET MVC и даже позволяет напрямую ссылаться на .js файл jquery на собственных CDN серверах Microsoft.

Страницы, сгенерированные ASP.NET MVC, не содержат никаких данных View State, поэтому они могут быть в сотни килобайт меньше, чем обычные страницы, созданные при помощи ASP.NET Web Forms. Несмотря на современную широкополосную связь и быстрые подключения, эта экономия пропускной способности до сих пор чрезвычайно притягательна для конечных пользователей.

Как Ruby on Rails, ASP.NET MVC работает в гармонии с HTTP. Вы полностью контролируете запросы, проходящие между браузером и сервером, поэтому вы можете подогнать настройки под себя, на сколько вам это нравится. AJAX сделан просто, и нет никакого автоматического обратного вмешательства в состояния на стороне клиента. Любой разработчик, который в первую очередь фокусируется на веб программировании, почти наверняка посчитает это освобождением и будет наслаждаться рабочим процессом.

## Тестируемость

Архитектура MVC дает вам отличную возможность создавать ваше приложение таким, чтобы его можно было легко сопровождать и тестировать, потому что вы, естественно, захотите разделить логические блоки приложения по независимым частям программного обеспечения. Тем не менее, создатели ASP.NET MVC на этом не остановились. Для поддержки модульного тестирования они приняли компоненто-ориентированный дизайн фреймворка и убедились, что каждая отдельная часть построена так, чтобы отвечать требованиям модульного тестирования.

Они добавили мастера (визарды) Visual Studio для создания начальных проектов модульного тестирования от вашего имени, которые интегрированы с инструментами модульного тестирования с открытым исходным кодом, такими как NUnit и XUnit, а также собственным

MSTest Microsoft. Даже если вы никогда не писали модульных тестов, вам будет легко в этом разобраться.

В этой книге вы увидите примеры того, как писать чистые, простые модульные тесты для ASP.NET MVC контроллеров и действий, с поддержкой фальшивых (fake) и фиктивных (mock) реализаций фреймворк компонентов для моделирования любых сценариев, используя различные стратегии тестирования.

Тестируемость – это не только вопрос модульного тестирования. ASP.NET MVC приложения также хорошо работают с инструментами автоматического тестирования. Вы можете написать тестовые скрипты, которые имитируют взаимодействие с пользователем, без необходимости гадать, какие структуры HTML элементов, CSS классы или ID будет генерировать фреймворк, и вам не придется беспокоиться о структуре, если она вдруг неожиданно изменится.

## **Мощная система маршрутизации (роутинга)**

Стиль ссылок изменился, поскольку технология веб приложений улучшилась. Такие ссылки, как эта:

```
/App_v2/User/Page.aspx?action=show%20prop&prop_id=82742
```

можно встретить довольно редко. Теперь они заменены более простым и чистым форматом:

```
/to-rent/chicago/2303-silver-street
```

Есть несколько веских причин для заботы о структуре URL. Во-первых, поисковые системы придают значительный вес ключевым словам, находящимся в URL. Поиск "аренда в Чикаго" (rent in Chicago) имеет гораздо больше шансов с простым URL. Во-вторых, многим пользователям Интернета теперь хватит навыков и знаний, чтобы понять URL, и оценить возможности навигации, набрав его в адресной строке своего браузера. В-третьих, когда кто-то понимает структуру URL, он, скорее всего, будет ссылаться именно на него, поделится этой ссылкой с другом или даже продиктует ее вслух по телефону. В-четвертых, такая ссылка не предоставляет технические подробности, папки, имена файлов и структуру приложения на весь общественный Интернет, так что вы можете изменить внутреннюю реализацию, не нарушая ссылки.

В более ранних фреймворках было сложно реализовать чистые ссылки, но ASP.NET MVC использует возможность `System.Web.Routing`, которая по умолчанию создает чистые URL-адреса. Теперь вы можете контролировать схему ссылок и ее связь и отношение к приложению, то есть вы свободны в создании шаблона URL-адресов, которые являются значимыми и полезными для пользователей, без необходимости соответствовать предопределенному шаблону. И, конечно, это означает, что вы можете легко определить современную URL схему в стиле REST, если захотите. Более подробную информацию о маршрутизации и лучших способах создания чистых URL вы найдете в главах 13 и 14.

## **Возможность разрабатывать в лучших сегментах платформы ASP.NET**

Существующая платформа Microsoft ASP.NET предоставляет зрелый, хорошо зарекомендовавший себя набор компонентов и средства для разработки эффективных и действенных веб приложений.

Во-первых, и что наиболее очевидно, поскольку ASP.NET MVC основан на .NET платформе, у вас есть возможность писать код на любом .NET языке и иметь доступ к тем же API функциям, не

только к MVC, но и к обширной .NET библиотеке классов и огромной экосистеме сторонних .NET библиотек.

Во-вторых, готовые возможности платформы ASP.NET, такие как мастер-страницы, аутентификация, роли, профили и интернационализация, могут уменьшить количество кода, который нужно писать и поддерживать для любых веб приложений, и эти функции так же эффективны при использовании в MVC Framework, как и в классических проектах Web Forms. Вы можете заново использовать некоторые встроенные серверные элементы управления Web Forms, а также свои собственные элементы управления из предыдущих проектов ASP.NET в приложениях ASP.NET MVC (если они не зависят от некоторых конкретных возможностей Web Forms, таких как View State).

## **Современный API**

С момента своего создания в 2002 году .NET платформа Microsoft неумолимо развивалась, поддерживая и даже определяя аспекты современного программирования.

ASP.NET MVC 4 был создан для .NET 4.5, поэтому его API в полной мере принял преимущества самых современных языков и технологий, в том числе ключевое слово await, методы расширений, лямбда-выражения, анонимные и динамические типы и LINQ (Language Integrated Query). Многие из методов API MVC Framework и паттерны кодирования следуют более чистым, более выразительным композициям, чем это было возможно на более ранних платформах.

## **ASP.NET MVC имеет открытый исходный код**

В отличие от предыдущих платформ веб-разработки от Microsoft, вы можете загрузить исходный код для ASP.NET MVC и даже изменить и скомпилировать собственную версию. Это имеет неоценимое значение, когда ваша отладка касается системы компонентов, и вы хотите зайти в код (и даже прочитать комментарии программистов-создателей). Это также полезно, если вы создаете «продвинутые» компоненты и хотите посмотреть, какие возможности существуют для их развития или как действительно работают встроенные компоненты.

Кроме того, эта возможность хороша в том случае, если вам не нравится, как что-то работает, если вы нашли ошибку или если вы просто хотите получить доступ к чему-то такому, что в противном случае недоступно, потому что вы можете просто изменить это самостоятельно. Тем не менее, вам нужно будет отслеживать изменения и повторять их, если вы перейдете на более новую версию платформы. ASP.NET MVC лицензирован Microsoft Public License (Ms-PL, <http://www.opensource.org/licenses/ms-pl.html>), а Open Source Initiative (OSI)-утвердил открытую лицензию. Это означает, что вы можете менять исходный код, разворачивать его и даже распространять ваши изменения публично как вами созданный проект. Вы можете скачать исходный код MVC на <http://aspnetwebstack.codeplex.com>.

## **Кто должен использовать ASP.NET MVC**

Как и с любой новой технологией, факт существования ASP.NET MVC не является веской причиной для того, чтобы работать с ним. Здесь мы хотим сравнить MVC Framework с наиболее очевидными альтернативами. Мы постарались быть максимально объективными для двух человек, которые пишет книгу о MVC Framework, но мы знаем, что есть предел нашей объективности. В следующих разделах представлены сравнения с другими технологиями. При выборе фреймворка для разработки веб приложения, вы также должны рассмотреть навыки вашей команды, принять

во внимание работу, связанную с переносом существующих проектов, и ваше понимание существующих технологий.

## Сравнение с ASP.NET Web Forms

Мы уже подробно описали слабые стороны и ограничения ASP.NET Web Forms и то, как ASP.NET MVC преодолевает многие из этих сложностей. Однако, это не обозначает, что технология Web Forms мертва. Microsoft неоднократно заявлял, что обе технологии активно развиваются и активно поддерживаются, и что нет никаких планов отказаться от Web Forms. В некотором смысле, выбор между этими двумя технологиями является вашей философией программиста. Давайте их сравним:

- Web Forms работает с представлением, как с сохраняющим состояния (являющимся stateful), добавляя абстрактный слой для HTTP и HTML. Используя View State и функции обратного вызова (postback) для создания эффекта сохранения состояния (statefulness). Благодаря этому возможна разработка drag-and-drop в стиле Windows Forms, то есть вы вставляете UI виджеты в шаблоны и заполняете кодом обработчики событий.
- MVC же включает в себя истинную природу HTTP, работая с ним, а не борясь против. MVC Framework требует понимания, как веб приложения работают на самом деле. Благодаря этому вы ощутите, что представляет собой простой, мощный, современный подход к написанию веб приложений с чистым кодом, который легче расширить и поддерживать в течение долгого времени без странных осложнений и болезненных ограничений.

Есть, конечно, случаи, когда Web Forms, по крайней мере, можно так же хорошо, и, наверное, даже лучше применять, чем MVC. Очевидным примером являются небольшие приложения для интранет, которые в основном привязаны непосредственно к таблицам базы данных. Сильные стороны drag-and-drop программирования Web Forms могут перевесить его слабые стороны, когда вам не нужно беспокоиться о пропускной способности или поисковой оптимизации.

Если же вы пишете приложения для Интернета или большие интранет приложения, вас привлечет хорошая пропускная способность, лучшая совместимость с браузерами и более продуманная поддержка автоматизированного тестирования, что и предлагает MVC.

## Переход от Web Forms к MVC

Если у вас уже есть ASP.NET Web Forms проект, который вы планируете перевести на MVC, вам будет приятно узнать, что эти две технологии могут сосуществовать в одном приложении. Это дает возможность для постепенного переноса существующих приложений, особенно если приложение разбивается на слои с моделью предметной области или бизнес логики, которые ограничены от страниц Web Forms. В некоторых случаях, возможно, стоит даже намеренно разработать приложение, которые будет представлять собой гибрид двух технологий.

## Сравнение с Ruby on Rails

Rails стал эталоном, с которым сравниваются другие веб платформы. Для разработчиков и компаний, которые живут в мире Microsoft .NET, гораздо легче принять и обучиться ASP.NET MVC, в то время как разработчики и компании, которые работают с Python или Ruby на Linux или Mac OS X найдут более легкий путь к Rails. Маловероятно, что вы захотите перейти от Rails на ASP.NET MVC или наоборот. Между этими двумя технологиями существуют некоторые реальные различия.

Rails представляет собой *целостную* платформу для разработки, что обозначает, что он работает с полным стеком, начиная от управления базой данных, через ORM, к обработке запросов с помощью контроллеров и действий и со встроенными средствами автоматизированного тестирования.

ASP.NET MVC Framework фокусируется на обработке веб запросов в MVC паттерне с помощью контроллеров и действий. Он не имеет встроенного ORM инструмента, встроенных средств автоматизированного тестирования или системой управления переносом баз данных. Это потому что у .NET платформы уже есть огромный выбор возможностей для выполнения этих функций, и вы можете использовать любую из них. Например, если вы ищете инструмент ORM, вы можете использовать NHibernate, Subsonic, Microsoft Entity Framework или одно из многих других зрелых имеющихся решений. Такова роскошь .NET платформы, хотя это не обозначает, что данные компоненты не так тесно интегрированы в ASP.NET MVC, как эквиваленты в Rails.

## Сравнение с MonoRail

MonoRail является более ранней платформой для веб приложений .NET MVC, созданной в рамках проекта с открытым исходным кодом Castle, и эта платформа развивается с 2003 года. Во многих отношениях MonoRail выступал в качестве прототипа для ASP.NET MVC. MonoRail продемонстрировали, как подобная Rails MVC архитектура может быть выстроена для ASP.NET и предоставил модели, методы и терминологию, которые используются в реализации Microsoft.

Мы не видим в MonoRail серьезного конкурента. Это, пожалуй, самая популярная .NET платформа для веб приложений, созданная за пределами Redmond, и в свое время она получила широкое распространение. Тем не менее, с момента запуска ASP.NET MVC, о проекте MonoRail слышно нечасто. Импульс энтузиазма и новаторства в мире .NET веб разработки в настоящее время сосредоточен на ASP.NET MVC.

## Что нового в ASP.NET MVC 4?

С версией 4 MVC Framework появился целый ряд улучшений по сравнению с версией 3. Есть несколько новых важных функций, таких как поддержка *Web API* приложений (о чем мы поговорим в главе 25), поддержка мобильных устройств (глава 24) и некоторые полезные методы оптимизации отправки содержимого клиентам (глава 24).

Кроме того, существует множество небольших улучшений, таких как упрощенный синтаксис для Razor, более хорошо организованная система предоставления информации об основной конфигурации в MVC приложениях и некоторые новые опции шаблона для проектов Visual Studio MVC.

## Резюме

В этой главе мы описали, как веб разработка с огромной скоростью развивалась от первых CGI исполняемых до новейших высокопроизводительных, соответствующих стандартам платформ. Мы проанализировали сильные и слабые стороны и ограничения ASP.NET Web Forms, основной веб платформы Microsoft с 2002 года, и рассказали об изменениях в индустрии веб разработки, которые заставили Microsoft в ответ придумать что-то новое.

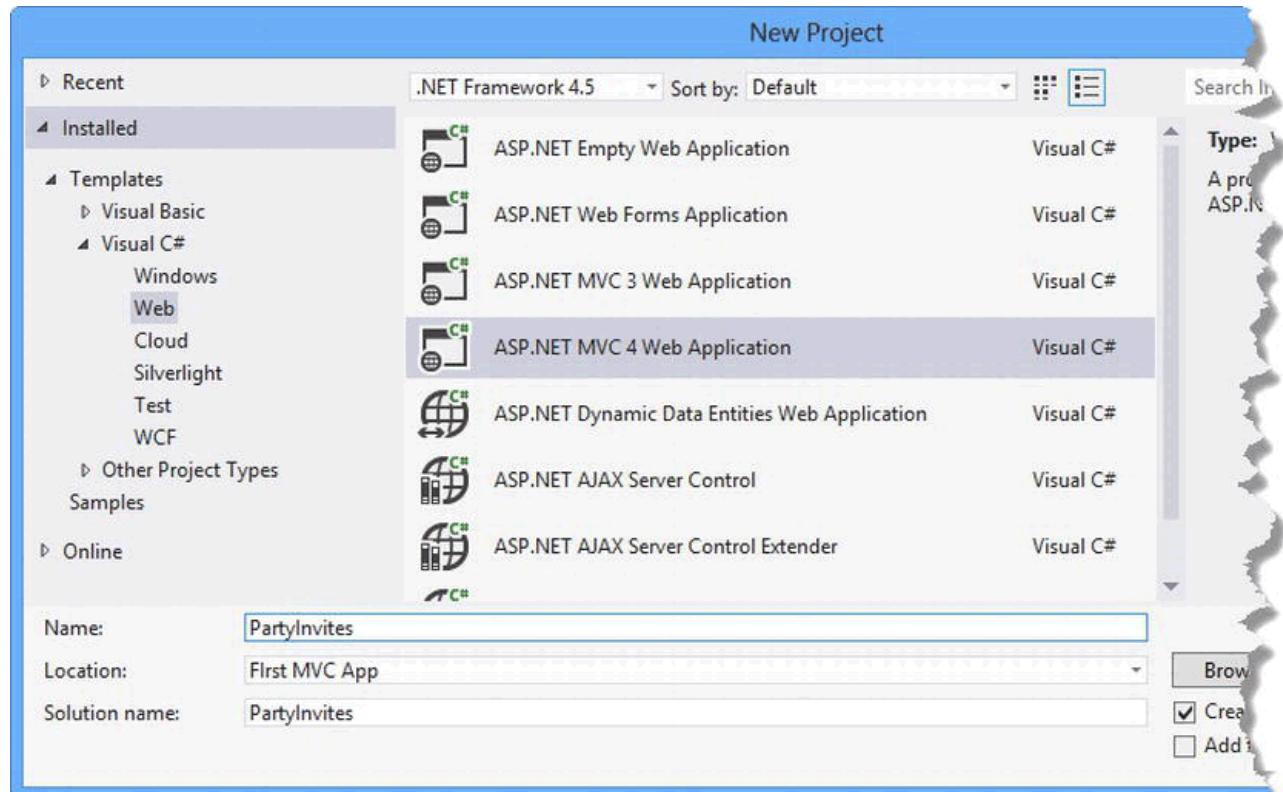
Вы увидели, как платформа ASP.NET MVC устраниет слабые стороны ASP.NET Web Forms, и как ее современный дизайн обеспечивает преимущества для разработчиков, которые хотят писать код высокого качества.

В следующей главе вы увидите, как работает MVC Framework, изучая простые механизмы, которые предоставляют все эти преимущества. Когда вы приступите к главе 7, вы будете готовы к созданию реального приложения по электронной коммерции, построенного при помощи чистой архитектуры, с правильным разделением задач, автоматизированными тестами и красивой минимальной разметкой.

# Создание нового ASP.NET MVC проекта

Мы начнем с создания нового MVC проекта в Visual Studio. Выберите New Project из меню File, чтобы открыть диалоговое окно New Project. Если вы выберите в разделе Visual C# шаблоны Web, вы увидите, что одним из доступных типов проекта является ASP.NET MVC 4 Web Application. Выберите этот тип проекта, как показано на [рисунке 2-1](#).

Рисунок 2-1: Шаблон Visual Studio MVC 4 проекта

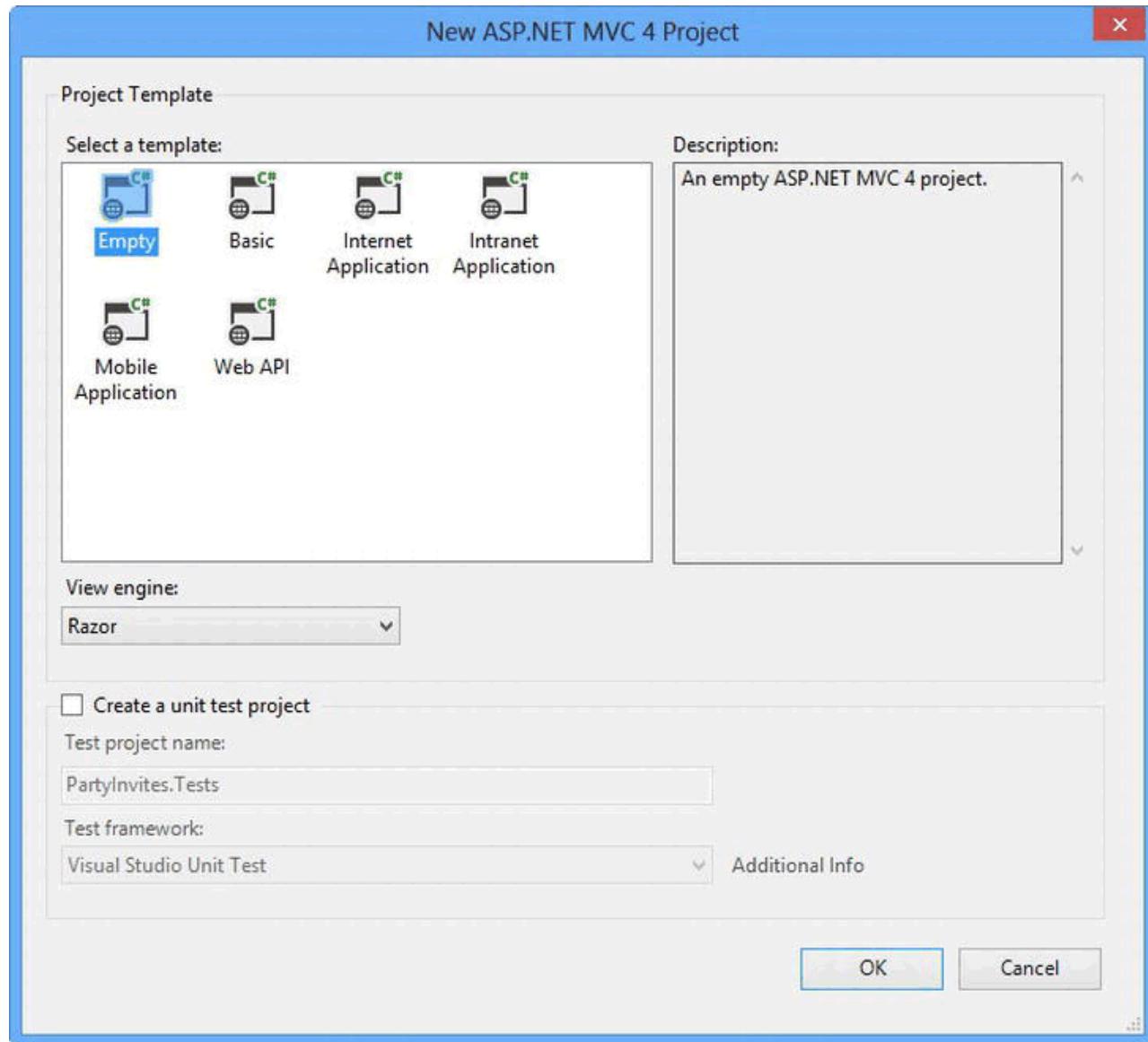


## Внимание

*Visual Studio 2012 включает в себя поддержку MVC 3, также как и MVC 4, и вы видите, что старые шаблоны доступны вместе с новыми. При создании нового проекта обратите на это внимание и выберите правильный*

Назовите новый проект PartyInvites и нажмите кнопку OK, чтобы продолжить. Вы увидите другое диалоговое окно, показанное на [рисунке 2-2](#), где вас попросят выбрать между тремя различными типами шаблонов MVC проекта.

Рисунок 2-2: Выбор типа MVC 4 проекта



Разные шаблоны MVC проектов создают проекты с разной базовой поддержкой таких функций, как аутентификация, навигация и стили. Мы не будем все усложнять в этой главе. Выберите вариант **Empty**, который создает проект с базовой структурой папок, но без файлов, необходимых для создания MVC приложений. Мы будем добавлять файлы, которые нам понадобятся, по мере прочтения главы, и каждый раз будем пояснять, что мы делаем.

Нажмите кнопку **OK**, чтобы создать новый проект.

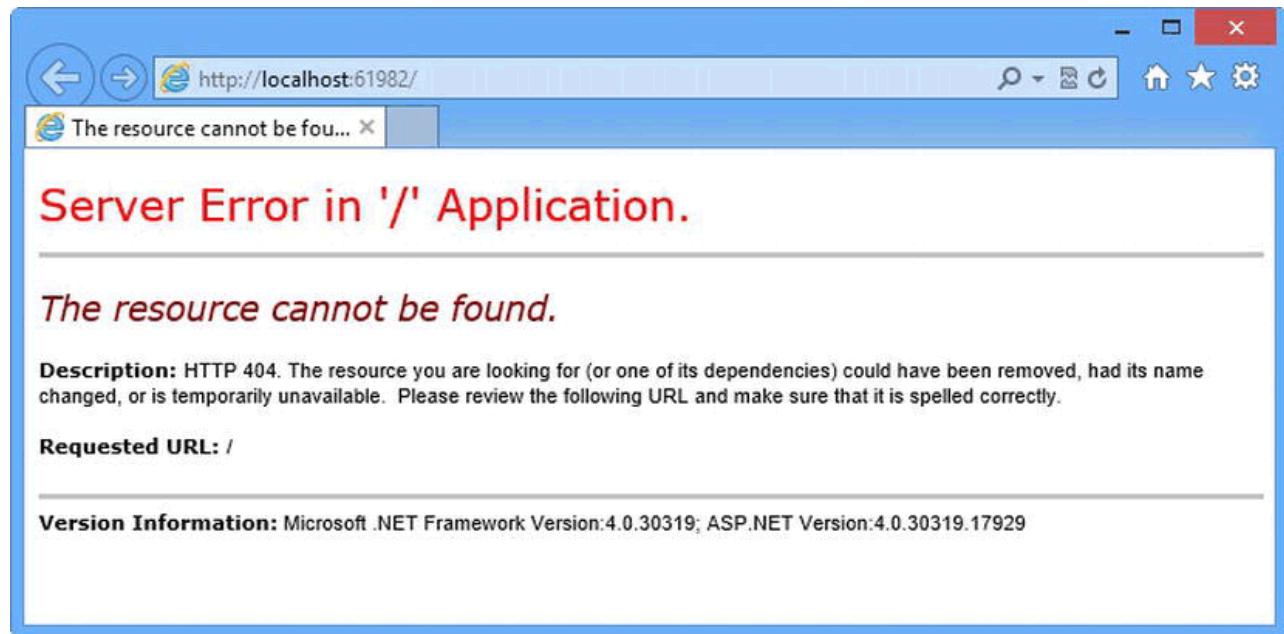
#### Примечание

*На рисунке 2-2 вы можете увидеть выпадающее меню, которое позволяет вам указать вид движка представления. В MVC 3 Microsoft представила новый и улучшенный вид движка, который называется **Razor**, и мы будем использовать Razor в этой книге. Мы рекомендуем вам сделать то же самое. Но если вы хотите использовать стандартный вид ASP.NET движка (известный как **ASPx**) – это ваш*

выбор. Мы расскажем все о Razor и о том, что делает движок представления, в главах 5 и 18.

Когда Visual Studio создаст проект, вы увидите файлы и папки, отображаемые в окне Solution Explorer. Это стандартная структура MVC 4 проекта. Вы можете попробовать запустить приложение, выбрав Start Debugging из меню Debug (если он попросит вас включить отладку, просто нажмите кнопку OK). Результат показан на [рисунке 2-3](#). Поскольку мы начали с пустого шаблона проекта, приложение ничего не содержит, так что мы получаем ошибку 404 Not Found.

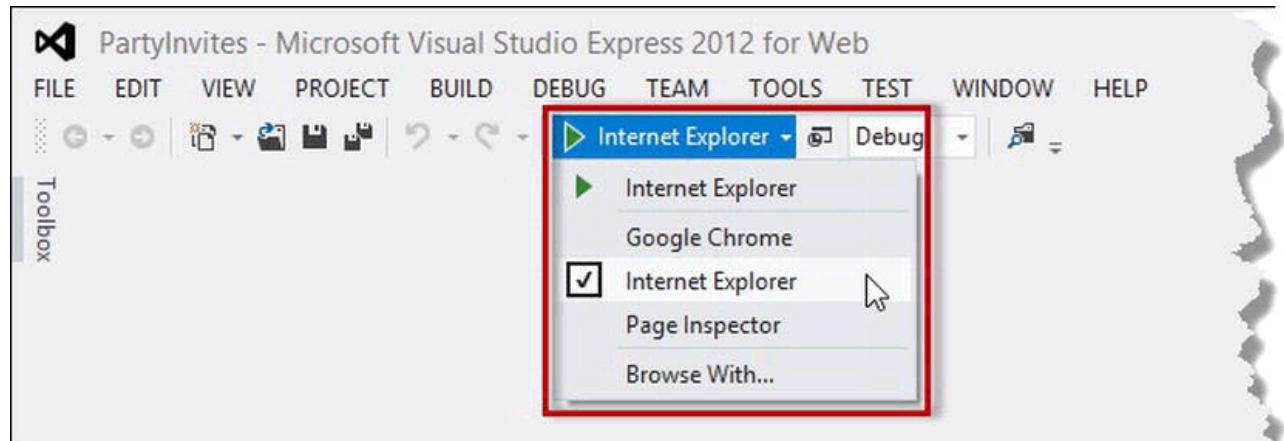
**Рисунок 2-3:** Попытка запустить пустой проект



Когда вы закончите, не забудьте остановить отладку, закрыв окно браузера, который показывает ошибку, или вернитесь к Visual Studio и выберите Stop Debugging в меню Debug.

Visual Studio открывает браузер для отображения проекта, и вы можете изменить браузер, который используется, в меню, показанном на [рисунке 2-4](#). Вы видите, что тут представлены Microsoft Internet Explorer и Google Chrome.

**Рисунок 2-4:** Смена браузера, который Visual Studio использует для запуска проекта



В этой книге мы будем использовать Internet Explorer 10. Все современные веб браузеры довольно хорошо на сегодняшний день, но мы будем работать с IE, потому что он установлен у многих людей.

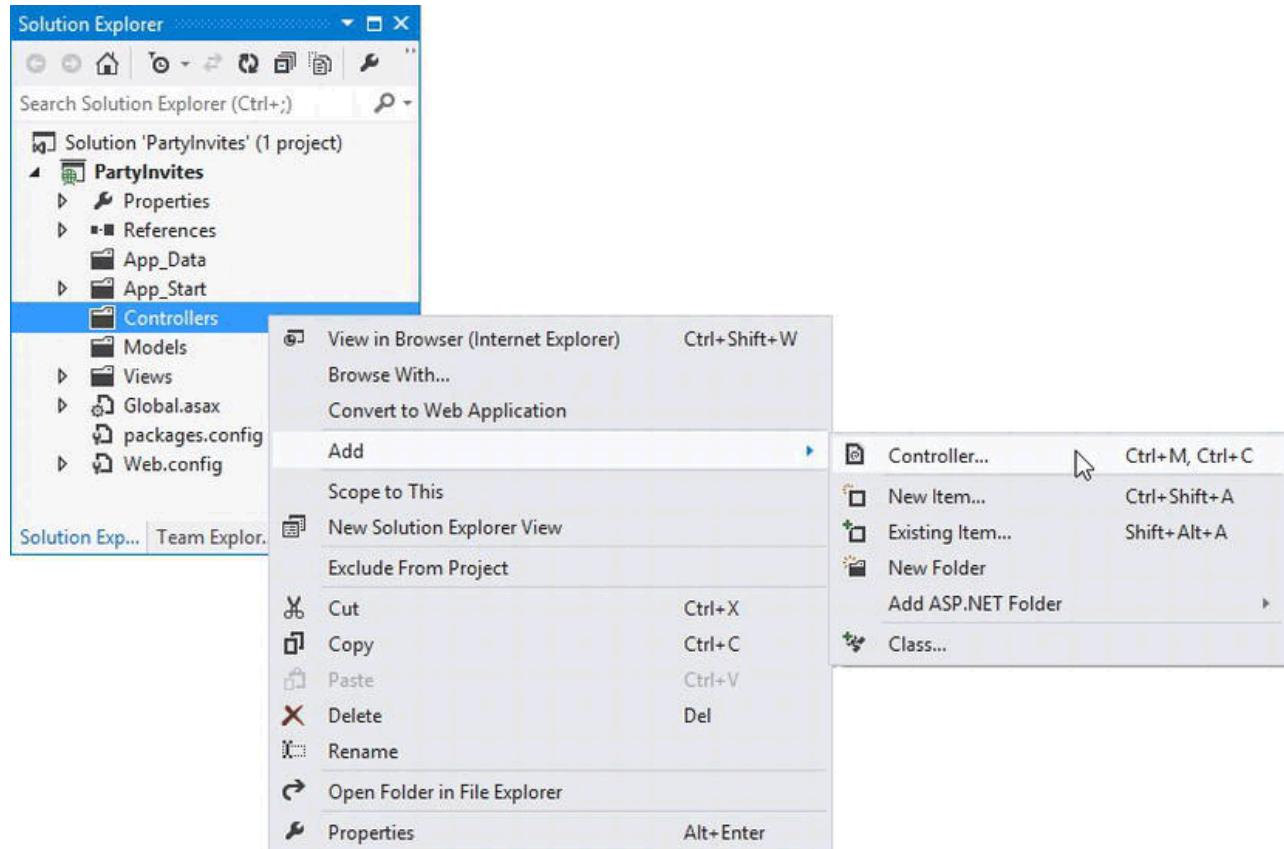
## Добавление первого контроллера

В архитектуре MVC входящие запросы обрабатываются *контроллерами*. В ASP.NET MVC контроллеры являются простыми C# классами (как правило, наследуются от `System.Web.Mvc.Controller`, встроенных во фреймворк базовых классов контроллеров). Каждый открытый метод в контроллере известен как *метод действия*, то есть вы можете вызвать его из Интернет через некоторые URL, чтобы выполнить действие. В MVC контроллеры находятся в папке под названием `Controllers`, которую Visual Studio создала для нас при создании проекта. Вам не нужно следить за этим и большинством других соглашений MVC, но мы рекомендуем вам сделать не в последнюю очередь потому, что это поможет вам разобраться в примерах, приведенных в данной книге.

Чтобы добавить контроллер в наш проект, просто щелкните правой кнопкой мыши по папке `Controllers` в окне Solution Explorer Visual Studio и затем выберите Add во всплывающем меню, как показано на [рисунке 2-5](#).

**Рисунок 2-5:** Добавление контроллера в MVC проект

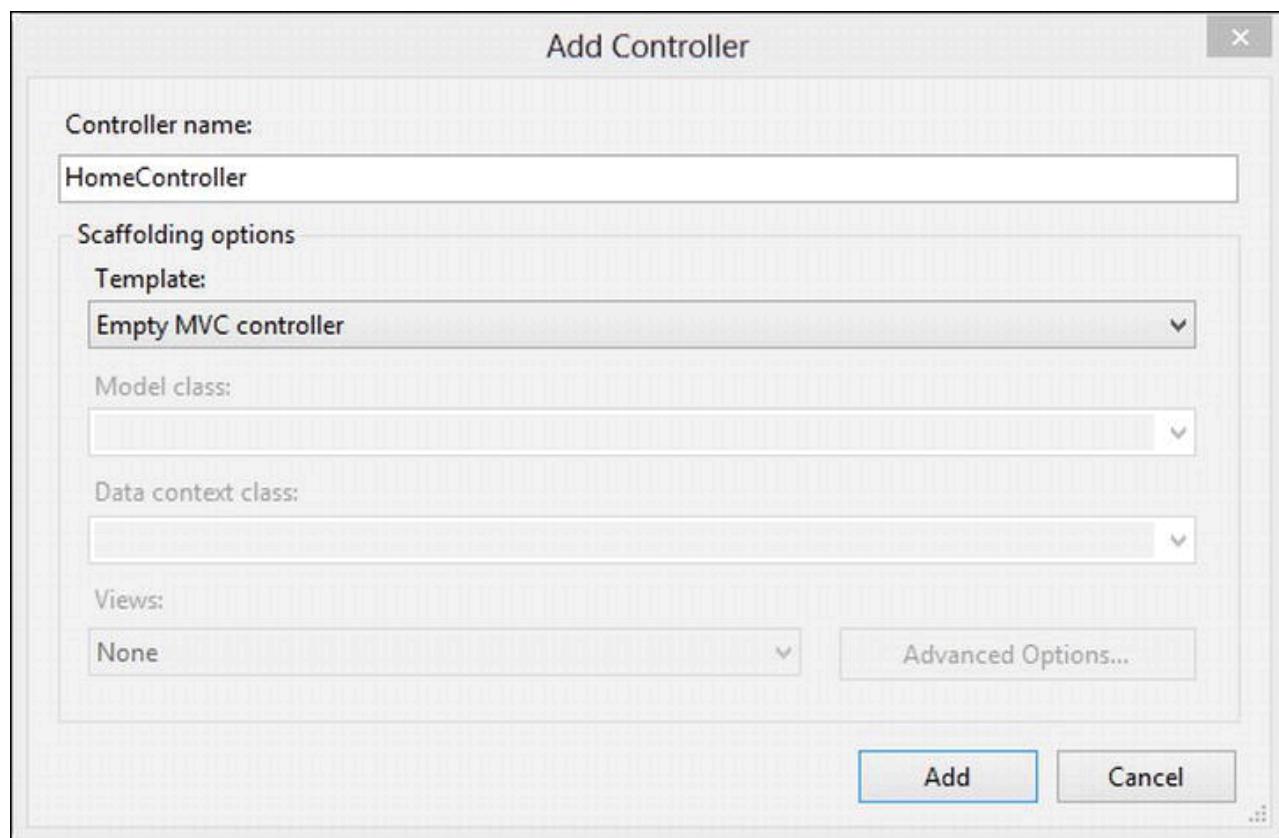
Когда



да появится диалоговое окно Add Controller, назовите контроллер `HomeController`, как показано

на [рисунке 2-6](#). Это еще одно соглашение: имена, которые мы даем контроллерам, должны быть описательными и заканчиваться Controller.

**Рисунок 2-6:** Называем контроллер



Раздел диалогового окна Scaffolding options позволяет нам создать контроллер с помощью шаблона с общими функциями. Мы не собираемся использовать эту возможность, поэтому убедитесь, что в меню Template выбрано Empty MVC controller, как показано на рисунке.

Нажмите кнопку Add, чтобы создать контроллер. Visual Studio создаст новый файл с C# кодом с названием HomeController.cs в папке Controllers и откроет его для редактирования. Мы показали контент по умолчанию, который Visual Studio помещает в классовый файл, в [листинге 2-1](#). Вы видите, что класс называется HomeController, и он является производным от System.Web.Mvc.Controller.

**Листинг 2-1:** Содержание по умолчанию класса HomeController

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
namespace PartyInvites.Controllers
{
    public class HomeController : Controller
```

```

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

```

Хороший способ начать работу с MVC – это сделать несколько простых изменений в классе контроллера. Измените код в файле `HomeController.cs` так, чтобы он соответствовал коду [листиングа 2-2](#). Мы выделили изменения, чтобы их было легче увидеть.

**Листинг 2-2:** Изменение класса `HomeController`

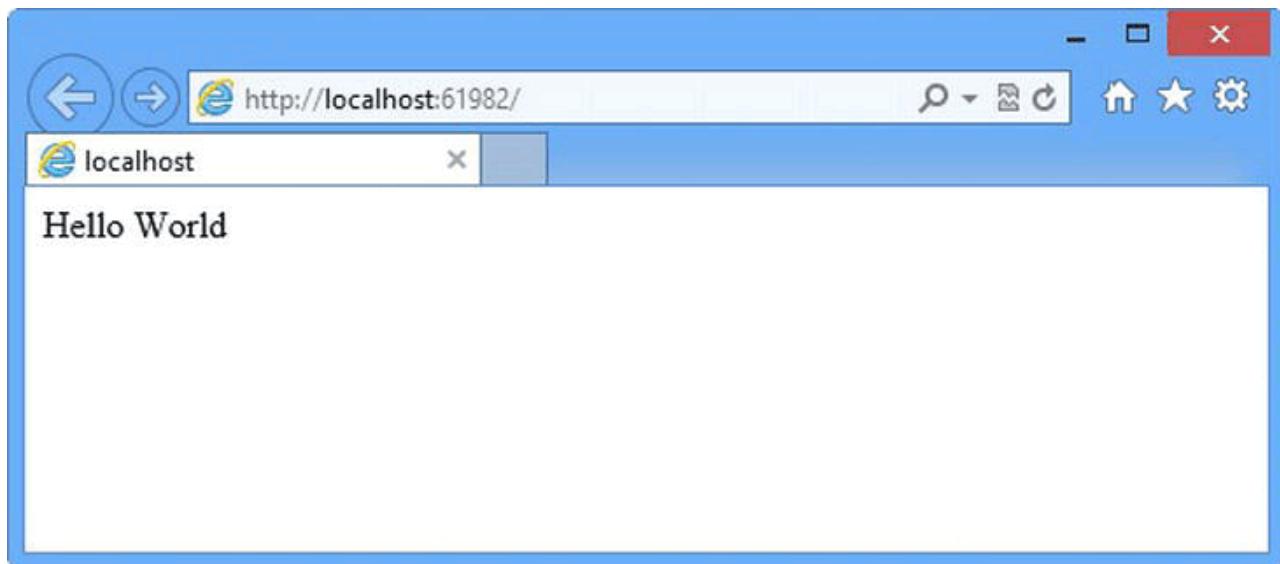
```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
namespace PartyInvites.Controllers
{
    public class HomeController : Controller
    {
        public string Index()
        {
            return "Hello World";
        }
    }
}

```

Мы не создали ничего захватывающего, но это хороший пример. Мы изменили метод действия (*action method*) `Index` таким образом, что он возвращает строку "Hello, world". Запустите проект еще раз, выбрав `Start Debugging` в Visual Studio меню `Debug`. Браузер отобразит результат метода действия `Index`, как показано на [рисунке 2-7](#).

**Рисунок 2-7:** Результат, возвращенный методом контроллера



## Роуты

Также как и модели, представления и контроллеры, MVC приложения используют систему маршрутизации (роутинговую систему) ASP.NET, которая решает, как URL-адреса картируют

конкретные контроллеры и действия. Когда Visual Studio создает MVC проект, она в начале добавляет некоторые роуты по умолчанию. Вы можете запросить любую из следующих ссылок, и они будут направлены на HomeController метод Index:

- /
- /Home
- /Home/Index

Поэтому когда браузер запрашивает `http://yoursite/` или `http://yoursite/Home`, он получает выходные данные HomeController метода Index. Вы можете попробовать сделать это самостоятельно, изменив URL в браузере. На данный момент, это будет `http://localhost:61982/`, за исключением того, что порт может быть другим. Если добавить в URL /Home или /Home/Index и обновить страницу, вы увидите тот же Hello World MVC приложения.

Это хороший пример пользы от MVC соглашений. В данном случае соглашение заключается в том, что у нас есть контроллер HomeController и что он будет отправной точкой для нашего MVC приложения. Роуты по умолчанию, которые Visual Studio создает для нового проекта, предполагают, что мы будем следовать этому соглашению. И так как мы *следовали* соглашению, мы получили поддержку для URL адресов из предыдущего списка.

Если бы мы *не* следовали соглашению, мы должны были бы изменить роуты до точки к тому контроллеру, который мы создали вместо этого. Для этого простого примера конфигурация по умолчанию это как раз то, что нам нужно.

#### *Совет*

*Вы можете просмотреть и отредактировать роутинговые настройки, открыв файл Global.asax.cs. В главе 7 вы будете создавать пользовательские записи маршрутизации, а в главах 13 и 14 вы узнаете гораздо больше о том, что может делать маршрутизация.*

## Представление (рендеринг) веб страниц

Результатом предыдущего примера не был HTML, это была просто строка "Hello World". Чтобы создать на запрос браузера HTML ответ, мы должны создать *представление*.

### Создание и обработка представления

Первое, что мы должны сделать, это изменить наш метод Index, как показано в [листе 2-3](#).

**Листинг 2-3:** Изменение контроллера для обработки представления

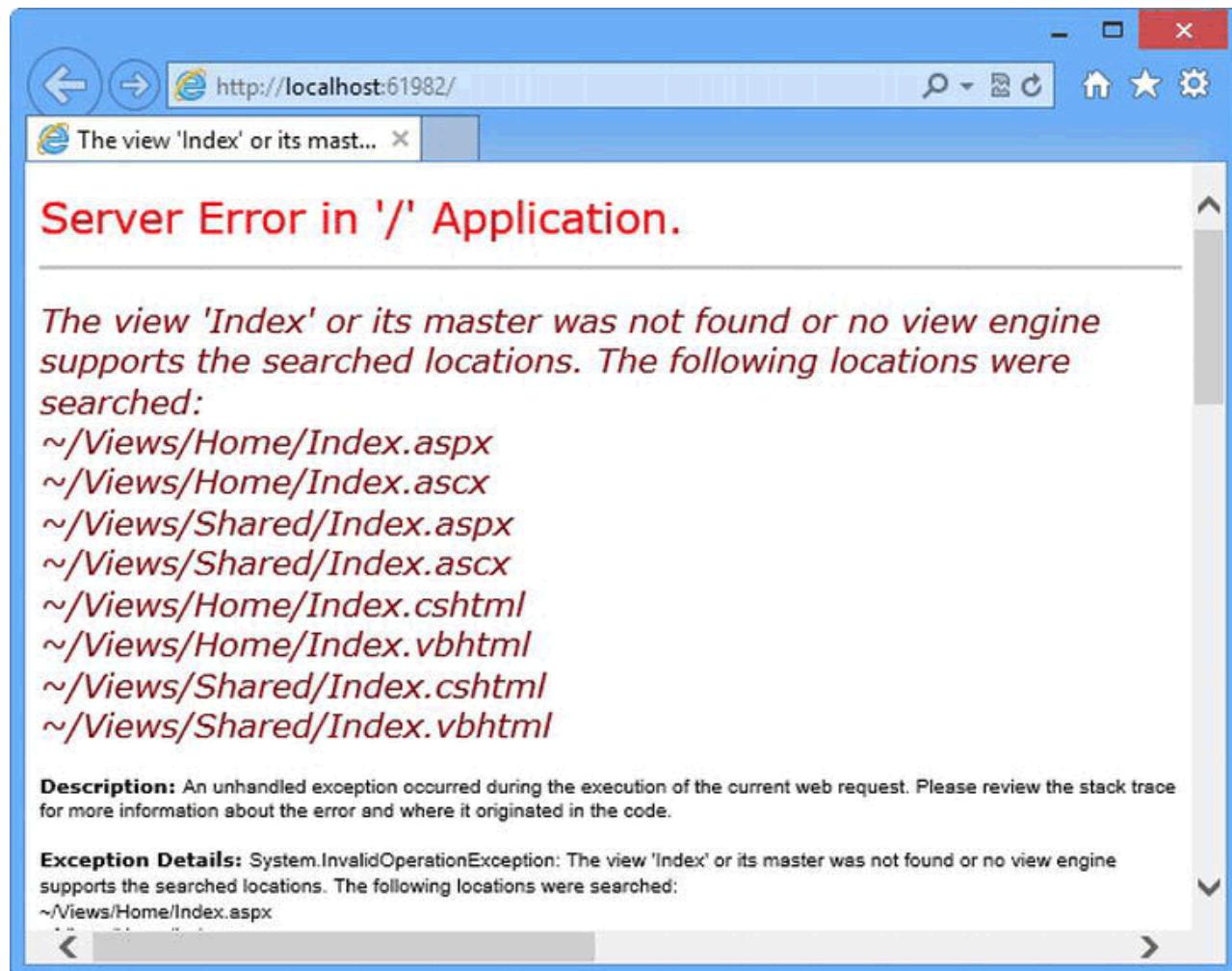
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
namespace PartyInvites.Controllers
{
    public class HomeController : Controller
    {
        public ViewResult Index()
```

```
    {
        return View();
    }
}
```

Изменения в [листинге 2-3](#) выделены жирным шрифтом. Когда мы возвращаемся к объекту ViewResult метода действия, мы поручаем MVC сделать представление. Мы создаем ViewResult, вызывая метод View без параметров. Это указывает MVC обрабатывать для метода действия представление *по умолчанию*.

Если вы сейчас запустите приложение, вы увидите, как MVC Framework пытается найти нужное представление по умолчанию, и это показано в сообщении об ошибке, которое представлено на [рисунке 2-8](#).

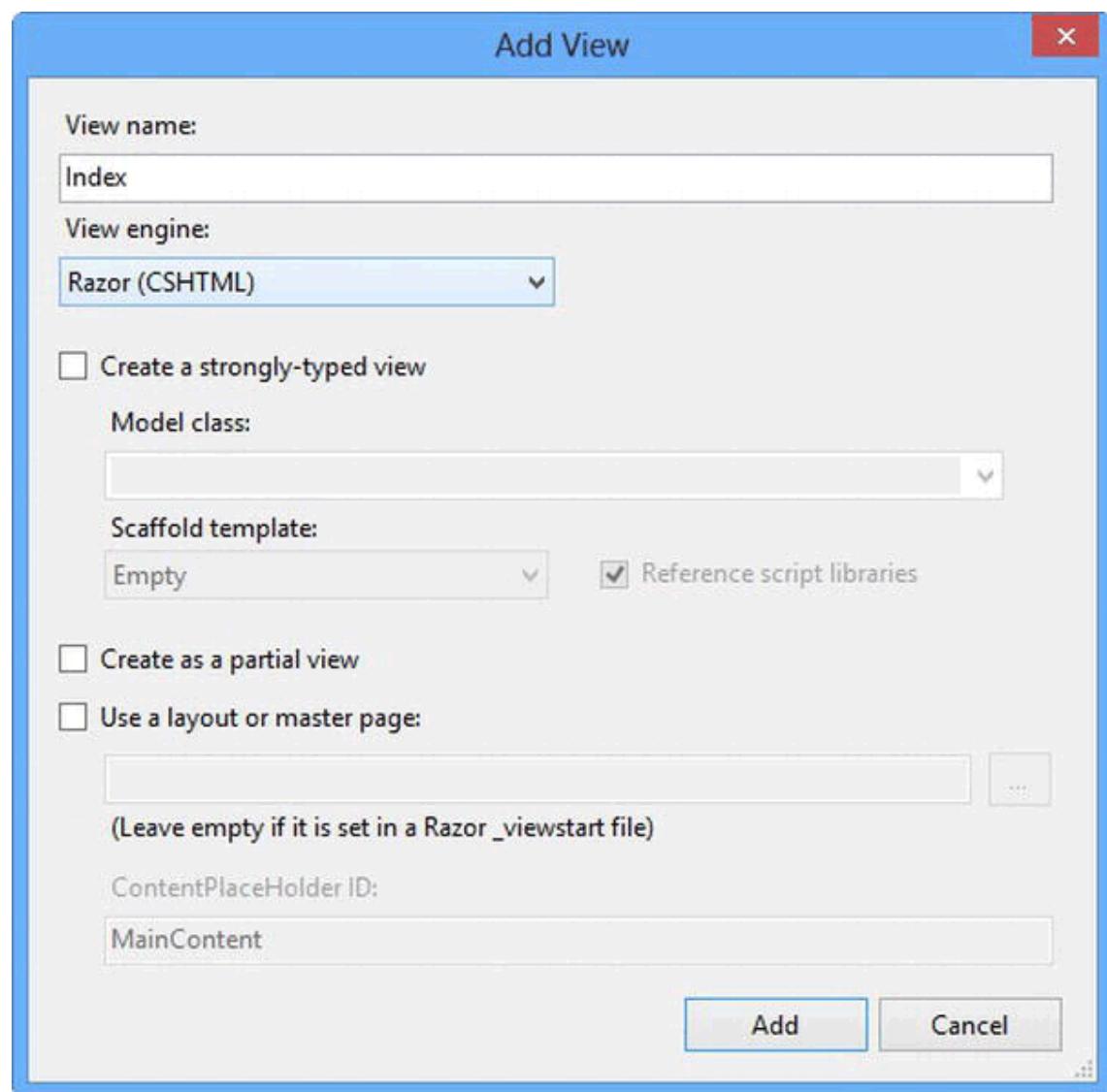
**Рисунок 2-8:** MVC Framework пытается найти представление по умолчанию



Это сообщение об ошибке весьма полезно. Оно объясняет не только то, что MVC не смог найти представление для нашего метода, но оно также показывает, где он искал. Это еще один хороший пример MVC соглашения: представления связаны с методами при помощи имен. Наш метод действия называется Index, а наш контроллер называется Home, и вы можете увидеть на [рисунке 2-8](#), что MVC пытается найти различные файлы в папке Views с таким именем.

Чтобы создать представление, остановите отладчик и щелкните правой кнопкой мыши по методу действия в кодовом файл HomeController.cs (либо по названию метода или внутри тела метода), а затем выберите из всплывающего меню Add View. Откроется диалоговое окно Add View, которое показано на [рисунке 2-9](#).

**Рисунок 2-9:** Диалоговое окно Add View



Снимите галочку с `Use a layout or master page`. В этом примере мы не используем макеты, но мы рассмотрим их в главе 7. Нажмите кнопку `Add`, и Visual Studio создаст новый файл с именем `Index.cshtml`, в папке `Views/Home`. Если вы посмотрите на сообщение об ошибке на [рисунке 2-8](#), вы увидите, что новый файл является одним из тех, что пытался найти MVC.

*Совет*

*Расширение файла `.cshtml` обозначает C# представление, которое будет обрабатываться Razor. Предыдущие версии MVC опирались на движок представлений ASPX, для которого файлы представления имели расширение `.aspx`.*

---

Visual Studio открывает `Index.cshtml` файл для редактирования. Вы видите, что этот файл содержит в основном HTML. Исключение составляет лишь та часть, которая выглядит следующим образом:

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

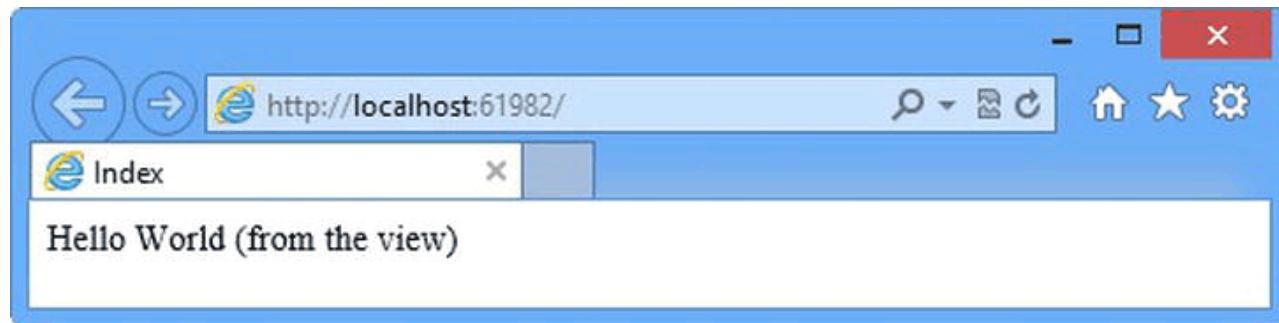
Данное выражение будет интерпретировано движком представления Razor. Это очень простой пример. Он просто говорит Razor, что мы решили не использовать мастер-страницу. На данный момент мы собираемся проигнорировать Razor и вернуться к нему позже. Дополните файл `Index.cshtml` тем выражением, которое выделено жирным шрифтом в [листинге 2-4](#).

**Листинг 2-4:** Добавление представления

```
@{  
    Layout = null;  
}  
<!DOCTYPE html>  
<html>  
<head>  
    <meta name="viewport" content="width=device-width" />  
    <title>Index</title>  
</head>  
<body>  
    <div>  
        Hello World (from the view)  
    </div>  
</body>  
</html>
```

С дополнением мы видим другое простое сообщение. Выберите `Start Debugging` в меню `Debug`, чтобы запустить приложение и проверить наше представление. Вы должны увидеть нечто похожее, что изображено на [рисунке 2-10](#).

**Рисунок 2-10:** Тестирование представления



Когда мы первый раз редактировали метод действия `Index`, он вернул строковое значение. Это обозначало, что MVC не сделал ничего, кроме как передал строковое значение браузеру. Теперь, когда метод `Index` возвращает `ViewResult`, мы поручаем MVC обработать представление и вернуть HTML. Мы не говорили MVC, какое представление должно быть использовано, поэтому он использовал соглашение по именованию, чтобы найти нужное автоматически. Соглашение состоит в том, что представление имеет название метода действия и содержится в папке, названной после контроллера: `~/Views/Home/Index.cshtml`.

Метод действия может возвращать другие результаты, кроме строк и объектов `ViewResult`. Например, если мы возвращаем `RedirectResult`, мы заставляем браузер перенаправиться на другой адрес. Если мы возвращаем `HttpUnauthorizedResult`, мы заставляем пользователя войти в систему (зарегистрироваться). Эти объекты известны как *результаты действия*, и все они происходят из класса `ActionResult`. Система результатов действий позволяет нам инкапсулировать и повторно использовать общие ответы на определенные действия. Мы расскажем вам о них и покажем на примерах далее в этой книге.

## Добавление динамических выходных данных

Весь смысл использования платформы для веб приложений заключается в построении и отображении *динамических* выходных данных. В MVC это работа контроллера – создать некоторые данные и передать их представлению, которое отвечает за то, чтобы представить их в виде HTML.

Одним из способов передачи данных от контроллера к представлению является использование объекта `ViewBag`, который является членом базового класса `Controller`. `ViewBag` – это динамический объект, которому можно присвоить произвольные свойства, что делает эти значения доступными для любого представления, которое будет с ними дальше работать. В [листинге 2-5](#) показана передача таким способом некоторых простых динамических данных в файл `HomeController.cs`.

### Листинг 2-5: Установка некоторых данных представления

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace PartyInvites.Controllers
{
    public class HomeController : Controller
```

```

{
    public ViewResult Index()
    {
        int hour = DateTime.Now.Hour;
        ViewBag.Greeting = hour < 12 ? "Good Morning" : "Good Afternoon";
        return View();
    }
}
}

```

Мы передали данные для представления, когда мы присвоили значение свойству `ViewBag.Greeting`. `ViewBag` является примером динамического объекта, а свойство `Greeting` не существовало до того момента, пока мы не присвоили ему значение. Это позволяет передать данные из контроллера в представление свободным и плавным образом, без необходимости досрочно определять классы.

Мы снова ссылаемся на свойство `ViewBag.Greeting` в представлении, чтобы получить значения данных, как показано в [листе 2-6](#), который демонстрирует изменения, что мы сделали в файле `Index.cshtml`.

#### **Листинг 2-6:** Получение значений данных `ViewBag`

```

@{
    Layout = null;
}

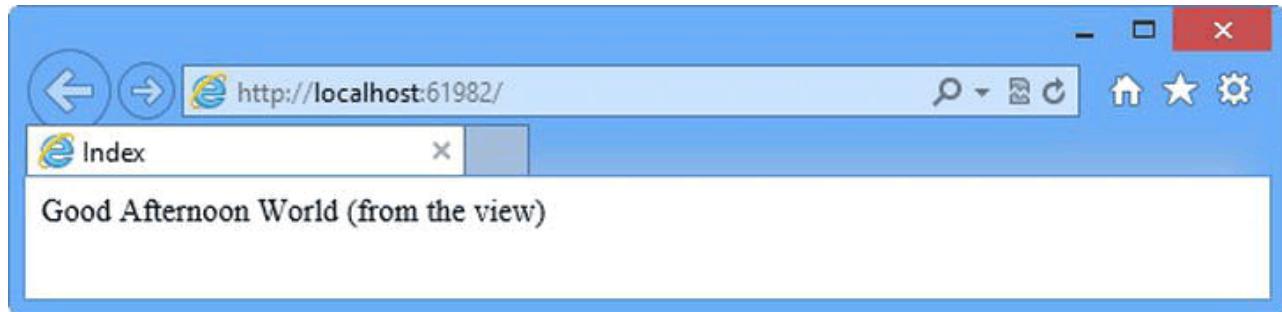
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
        @ViewBag.Greeting World (from the view)
    </div>
</body>
</html>

```

Дополнением в [листе 2-6](#) является выражение Razor. Когда мы вызываем метод `View` в методе контроллера `Index`, фреймворк MVC находит файл представления `Index.cshtml` и просит движок Razor разобрать (отпарсить) содержимое файла. Razor ищет выражение, как то, что мы добавили в листинг, и обрабатывает его. В этом примере обработка выражения обозначает вставку значения, которое мы присвоили свойству `ViewBag.Greeting` метода действия, в представление.

Там нет ничего особенного в имени свойства `Greeting`, вы можете заменить его любым именем свойства, и оно будет работать так же. Кроме того, вы можете передать несколько значений данных из контроллера в представление путем присвоения значений более чем одному свойству. Если мы запустим проект, мы увидим наши первые динамические выходные данные MVC, как показано на [рисунке 2-11](#).

#### **Рисунок 2-11:** Динамический ответ MVC



## Создание простого приложения по вводу данных

В остальной части этой главы мы рассмотрим несколько основных особенностей MVC путем создания простого приложения по вводу данных. В этом разделе мы собираемся немного ускориться. Наша цель заключается в демонстрации MVC в действии, поэтому мы опустим некоторые разъяснения о том, как все это работает. Но не волнуйтесь, мы подробно рассмотрим эти темы в следующих главах.

### Набросаем план

Давайте представим себе, что подруга решила провести вечеринку в канун Нового года, и что она попросила нас создать веб сайт, который позволит ее друзьям и знакомым принять приглашение с RSVP (подпись на приглашении, призывающая получателя дать ответ об участии в мероприятии). На сайте должно присутствовать следующее:

- Главная страница, где отображается информация о вечеринке
- Форма, которая может быть использована для RSVP
- Валидация RSVP формы, которая отобразит страницу с благодарностью
- Заполненный и отправленный ответ о согласии принять участие в вечеринке

В следующих разделах мы будем наращивать MVC проект, который мы создали в начале главы, и добавим эти возможности. Мы можем сделать первый пункт из списка, применив те знания то, которые мы получили ранее, то есть мы можем добавить HTML для наших существующих представлений, где будет дана подробная информация о вечеринке. В [листе 2-7](#) показаны дополнения, которые мы сделали в файле Views/Home/Index.cshtml.

### Листинг 2-7: Отображение информации о вечеринке

```
@{  
    Layout = null;  
}  
  
<!DOCTYPE html>  
<html>  
<head>  
    <meta name="viewport" content="width=device-width" />  
    <title>Index</title>  
</head>  
<body>  
    <div>  
        @ViewBag.Greeting  
        World (from the view)  
        <p>
```

```

We're going to have an exciting party.<br />
(To do: sell it better. Add pictures or something.)
</p>
</div>
</body>
</html>

```

Мы на правильном пути. Если вы запустите приложение, вы увидите информацию о вечеринке, ну, вернее, метку-заполнитель (placeholder) для этой информации, но вы можете уловить суть. Пример показан на [рисунке 2-12](#).

**Рисунок 2-12:** Добавление представления



## Проектирование модели данных

В MVC *M* обозначает *модель*, и это является самой важной частью приложения. Модель является представлением реальных объектов, процессов и правил, которые определяют объект, известный как *домен*, нашего приложения. Модель, которая часто упоминается как *доменная модель*, содержит C# объекты (известные как *доменные объекты*), которые составляют суть нашего приложения, и методы, которые позволяют нам манипулировать ими. Представления и контроллеры раскрывают домен клиентам в согласованном порядке, и хорошо продуманное MVC приложение начинается с хорошо продуманной модели, которая затем является координационным центром, когда мы добавляем контроллеры и представления.

Нам не нужна сложная модель для приложения `PartyInvites`, но мы создадим один доменный класс, которые мы назовем `GuestResponse`. Этот объект будет отвечать за хранение, проверку и подтверждение RSVP.

## Добавление класса модели

По MVC соглашению классы, которые составляют модель, помещаются в папку `Models`. Щелкните правой кнопкой мыши по `Models` в окне `Solution Explorer` и выберите `Add`, за которым следует `Class`, из всплывающего меню. Назовите файл `GuestResponse.cs` и нажмите кнопку `Add`, чтобы создать класс. Измените содержимое класса в соответствии с [листингом 2-8](#).

Совет

Если у вас нет пункта меню Class, то вы, вероятно, оставили работать отладчик (дебаггер) Visual Studio. Visual Studio ограничивает изменения, которые можно внести в проект, если приложение запущено.

---

### Листинг 2-8: Доменный класс GuestResponse

```
namespace PartyInvites.Models
{
    public class GuestResponse
    {
        public string Name { get; set; }
        public string Email { get; set; }
        public string Phone { get; set; }
        public bool? WillAttend { get; set; }
    }
}
```

### Совет

Вы, возможно, заметили, что свойство WillAttend имеет тип `bool?` (`Nullable<bool>`), то есть оно может быть `true`, `false` или `null`. Мы объясним это в разделе «Добавление валидации» далее в этой главе.

### Ссылка на метод действия

Одна из целей нашего приложения заключается во включении RSVP формы, поэтому нам нужно добавить ссылку на нее из нашего представления `Index.cshtml`, как показано в [листе 2-9](#).

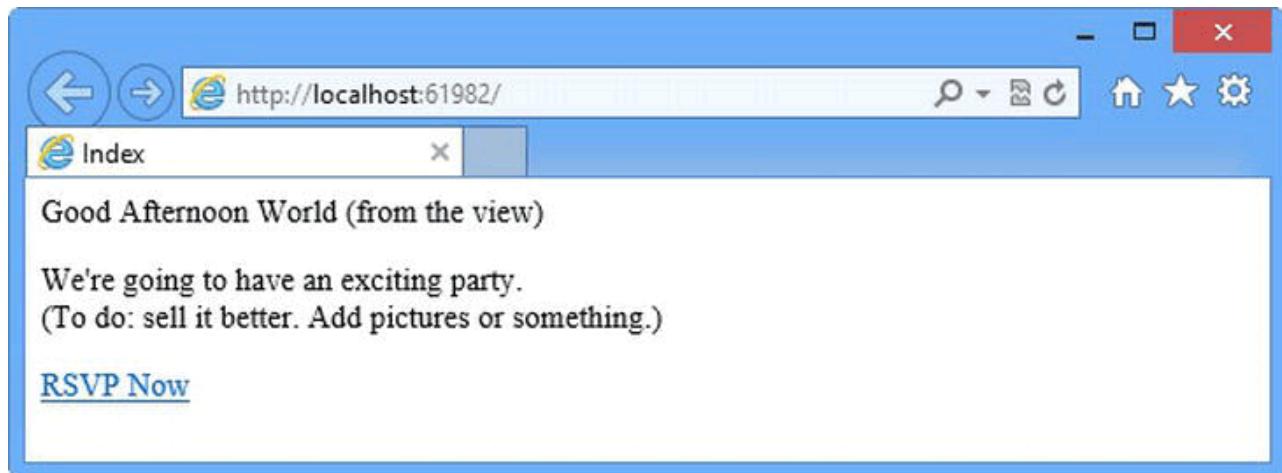
### Листинг 2-9: Добавление ссылки для RSVP формы

```
@{
    Layout = null;
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
        @ViewBag.Greeting World (from the view)
        <p>
            We're going to have an exciting party.<br />
            (To do: sell it better. Add pictures or something.)
        </p>
        @Html.ActionLink("RSVP Now", "RsvpForm")
    </div>
</body>
</html>
```

`Html.ActionLink` является *вспомогательным методом* HTML. MVC Framework разработан с набором встроенных вспомогательных методов, которые удобны для обработки HTML ссылок, текстовых вводных данных, флажков, выборок и даже пользовательских элементов управления. Метод `ActionLink` принимает два параметра: первый – это текст для отображения в ссылке, а второй – это выполняемое действие, когда пользователь нажимает на ссылку. Мы объясним вспомогательные методы HTML в главах 19-21. На [рисунке 2-13](#) показана ссылка, которую мы добавили.

**Рисунок 2-13:** Добавление в представление ссылки



Если вы наведете курсор мыши на ссылку в браузере, вы увидите, что ссылка указывает на `http://yourserver/Home/RsvpForm`. Метод `Html.ActionLink` проверил конфигурацию URL нашего приложения и определил, что `/Home/RsvpForm` является URL для действия `RsvpForm` контроллера `HomeController`. Обратите внимание, что в отличие от традиционных приложений ASP.NET, URL-адреса MVC не соответствуют физическим файлам. Каждый метод действия имеет свой URL, и MVC использует систему маршрутизации ASP.NET перевести эти URL в действия.

### Создание метода действия

Если вы нажмете на ссылку, то увидите ошибку `404 Not Found`. Это потому что пока еще мы не создали метод действия, который соответствует URL `/Home/RsvpForm`. Мы сделаем это путем добавления метода `RsvpForm` нашему классу `HomeController`, как показано в [листинге 2-10](#).

**Листинг 2-10:** Добавление в контроллер нового метода действия

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace PartyInvites.Controllers
{
    public class HomeController : Controller
    {
        public ViewResult Index()
        {
            int hour = DateTime.Now.Hour;
            ViewBag.Greeting = hour < 12 ? "Good Morning" : "Good Afternoon";
            return View();
        }

        public ViewResult RsvpForm()
        {
            return View();
        }
    }
}
```

### Добавление строго типизированного представления

Мы хотим добавить представление для нашего метода действия `RsvpForm`, но мы собираемся сделать кое-что больше: то есть создать *строго типизированное* представление. Стого типизированные представления предназначены для обработки определенного типа доменов. Если

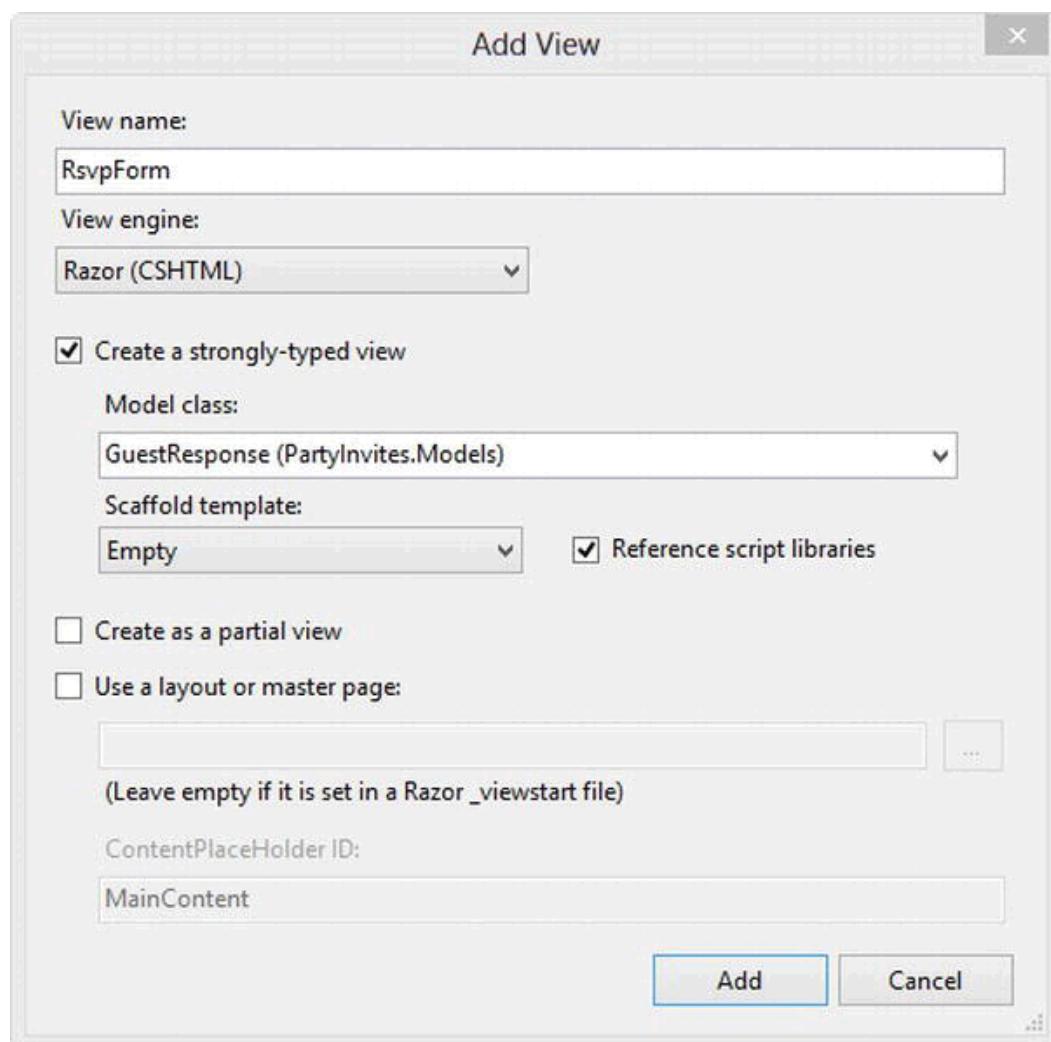
мы укажем тип, с которым мы хотим работать (в этом примере `GuestResponse`), MVC предоставит дополнительные возможности, чтобы упростить нам задачу.

#### Внимание

*Убедитесь, что перед работой ваш MVC проект скомпилирован. Если вы создали класс `GuestResponse`, но не скомпилировали его, MVC не сможет создать строго типизированное представление для данного типа. Чтобы скомпилировать приложение, выберите Build Solution в Visual Studio меню Build.*

Щелкните правой кнопкой мыши внутри метода действия `RsvpForm` и выберите для создания представления `Add View` из всплывающего меню. В диалоговом окне `Add View` поставьте галочку на `Create a strongly-typed view` и выберите опцию `GuestResponse` из выпадающего меню. Снимите флажок `Use a layout or master page` и убедитесь, что Razor выбран в качестве движка представления, и что опция `Scaffold template` установлена на `Empty`, как показано на [рисунке 2-14](#).

**Рисунок 2-14:** Добавление строго типизированного представления



Нажмите кнопку `Add`, и Visual Studio создаст новый файл с именем `RsvpForm.cshtml` и откроет его для редактирования. Вы можете увидеть первоначальное содержание в [листинге 2-12](#). Как вы

заметили, это другой HTML файл, но он содержит Razor выражение @model. Вы сейчас увидите, что это является ключом к строго типизированному представлению и возможностям, которые оно предлагает.

### Листинг 2-12: Начальное содержимое файла RsvpForm.cshtml

```
@model PartyInvites.Models.GuestResponse

@{
    Layout = null;
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>RsvpForm</title>
</head>
<body>
    <div>
    </div>
</body>
</html>
```

## Построение формы

Теперь, когда мы создали строго типизированное представление, мы можем выстроить содержание RsvpForm.cshtml, чтобы превратить его в HTML форму для редактирования GuestResponse объектов. Измените представление так, чтобы оно соответствовало [листиngу 2-13](#).

### Листинг 2-13: Создание представления формы

```
@model PartyInvites.Models.GuestResponse

@{
    Layout = null;
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>RsvpForm</title>
</head>
<body>
    @using (Html.BeginForm())
    {
        <p>Your name: @Html.TextBoxFor(x => x.Name) </p>
        <p>Your email: @Html.TextBoxFor(x => x.Email)</p>
        <p>Your phone: @Html.TextBoxFor(x => x.Phone)</p>
        <p>
            Will you attend?
            @Html.DropDownListFor(x => x.WillAttend, new[] {
                new SelectListItem() {Text = "Yes, I'll be there", Value =
bool.TrueString},
                new SelectListItem() {Text = "No, I can't come", Value = bool.FalseString}
            }, "Choose an option")
        </p>
        <input type="submit" value="Submit RSVP" />
    }
</body>
</html>
```

Для каждого свойства класса модели `GuestResponse` мы используем вспомогательный метод `HTML`, чтобы обработать подходящий HTML элемент управления `input`. Эти методы позволяют выбрать свойство, к которому относится элемент `input`, с помощью лямбда-выражения, например вот так:

```
@Html.TextBoxFor(x => x.Phone)
```

Вспомогательный метод HTML `TextBoxFor` генерирует HTML для элемента `input`, устанавливает параметр `type` на `text` и устанавливает атрибуты `id` и `name` на `Phone`, имя выбранного свойства доменного класса, вот так:

```
<input id="Phone" name="Phone" type="text" value="" />
```

Эта удобная функция работает, потому что наше представление `RsvpForm` строго типизировано, и мы сказали MVC, что `GuestResponse` это тот тип, который мы хотим обработать при помощи данного представления, поэтому вспомогательные методы HTML могут понять, какой тип данных мы хотим прочитать, с помощью выражения `@model`.

Не волнуйтесь, если вы не знакомы с лямбда-выражениями C#. Мы расскажем о них в главе 4, но в качестве альтернативы лямбда-выражениям вы можете обратиться к имени свойства типа модели как к строке, например, вот так:

```
@Html.TextBox("Email")
```

Мы считаем, что методика лямбда-выражений помогает нам не делать опечаток в имени свойства типа модели, потому что всплывает Visual Studio IntelliSense и позволяет нам выбрать свойство автоматически, как показано на [рисунке 2-15](#).

**Рисунок 2-15:** Visual Studio IntelliSense для лямбда-выражений во вспомогательных методах HTML



Другим удобным вспомогательным методом является `Html.BeginForm`, который генерирует элемент HTML формы, настроенный на обратную передачу данных методу действия. Поскольку мы не передали вспомогательному методу никаких параметров, он предполагает, что мы хотим

передать обратно тот же URL. Ловким трюком является то, чтобы обернуть это в C# выражение `using`, вот таким образом:

```
@using (Html.BeginForm()) {
    ...form contents go here...
}
```

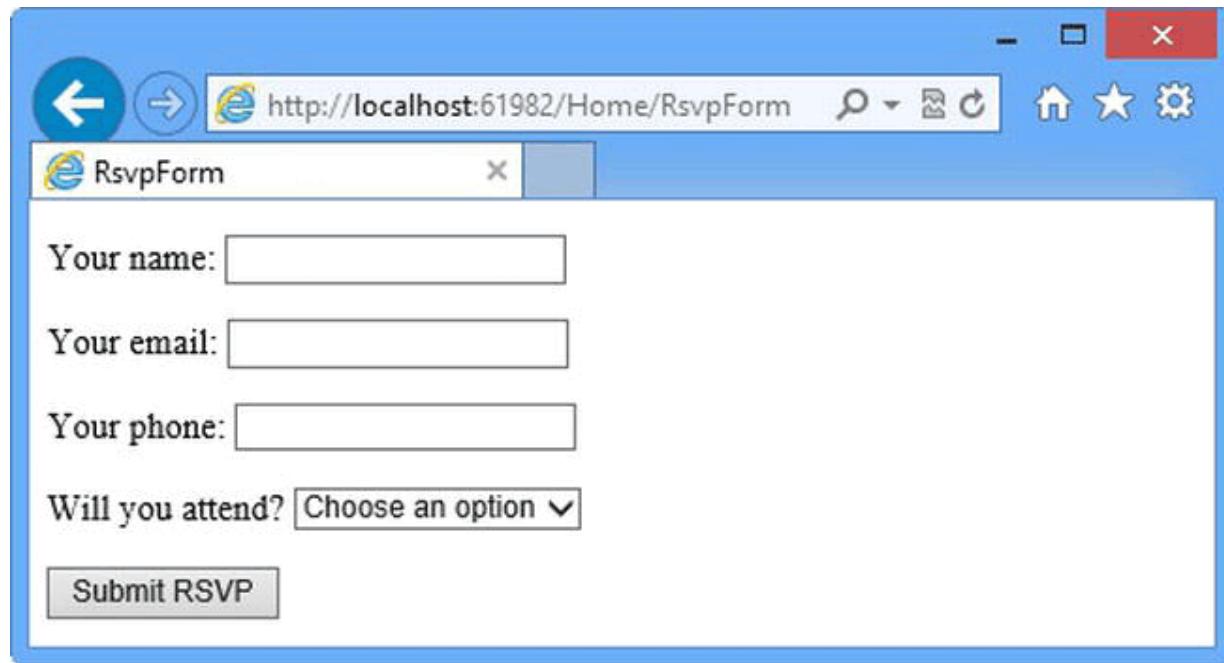
Обычно, когда оно применяется таким образом, выражение `using` гарантирует, что объект удаляется, когда выходит из области видимости. Оно широко используется для подключения к базе данных, например, чтобы убедиться, что она закрывается, как только запрос был выполнен. (Это применение ключевого слова `using` отличается от того, что касается области видимости класса).

Вместо удаления объекта, помощник `HtmlBeginForm` закрывает HTML элемент `form`, когда он выходит из области видимости. Это означает, что вспомогательный метод `Html.BeginForm` создает обе части элемента `form`, например:

```
<form action="/Home/RsvpForm" method="post">
    ...form contents go here...
</form>
```

Не волнуйтесь, если вы не знакомы с удалением объектов в C#. Наша цель на данный момент состоит в том, чтобы показать, как создать форму с помощью вспомогательного метода HTML. Вы можете видеть форму в представлении `RsvpForm`, когда вы запустите приложение и нажмете ссылку `RSVP Now`. На [рисунке 2-16](#) показан результат.

**Рисунок 2-16:** Представление RsvpForm



#### Примечание

Это не книга о CSS или веб дизайне. По большей части мы будем создавать примеры, внешний вид которых может быть описан как устаревший (хотя мы предпочитаем термин классический, в котором чувствует меньше пренебрежения). MVC представления генерируют очень чистый и простой HTML, и

*вы можете полностью управлять версткой элементов и классов, к которым они принадлежат, поэтому у вас не будет проблем с использованием дизайнерских средств или готовых шаблонов, чтобы сделать ваш MVC проект красивым.*

---

## Обработка форм

Мы не сказали MVC, что мы хотим сделать, когда форма отправляется на сервер. На данный момент нажатие на кнопку `Submit RSVP` просто удаляет любые значения, которые вы ввели в форму. Это потому что форма отправляется обратно методу действия `RsvpForm` в контроллере `Home`, который просто говорит MVC обработать представление еще раз.

### *Примечание*

*Вы можете быть удивлены тем фактом, что входные данные теряются, когда представление снова обрабатывается. Если это так, то вы, вероятно, разрабатывали приложения при помощи ASP.NET Web Forms, который автоматически сохраняет данные в этой ситуации. Мы покажем вам, как добиться такого же результата с MVC в ближайшее время.*

---

Чтобы получить и обработать отправленные данные формы, мы собираемся сделать умную и классную вещь. Мы добавим второй метод действия `RsvpForm` для того, чтобы создать следующее:

- *Метод, который отвечает на HTTP GET запросы:* GET запрос является тем, с чем браузер имеет дело после каждого клика по ссылке. Этот вариант действий будет отвечать за отображение начальной пустой формы, когда кто-то первый раз посетит `/Home/RsvpForm`.
- *Метод, который отвечает на HTTP POST запросы:* По умолчанию, формы, обрабатываемые при помощи `Html.BeginForm()`, отправляются браузером как POST запросы. Этот вариант действий будет отвечать за получение отправленных данных и решать, что с ними делать.

Обработка GET и POST запросов в отдельных методах C# помогает сохранить наш код опрятным, так как оба метода имеют различные обязанности. Оба метода действия вызываются одним и тем же URL, но MVC гарантирует, что будет вызван соответствующий метод в зависимости от того, имеем ли мы дело с GET или с POST запросом. В [листинге 2-14](#) показаны изменения, которые мы должны сделать в классе `HomeController`.

**Листинг 2-14:** Добавление метода действия для поддержки POST запросов

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using PartyInvites.Models;

namespace PartyInvites.Controllers
{
    public class HomeController : Controller
    {
        public ViewResult Index()
        {
            int hour = DateTime.Now.Hour;
            ViewBag.Greeting = hour < 12 ? "Good Morning" : "Good Afternoon";
            return View();
        }
    }
}
```

```
}

[HttpGet]
public ViewResult RsvpForm()
{
    return View();
}

[HttpPost]
public ViewResult RsvpForm(GuestResponse guestResponse)
{
    // TODO: Email response to the party organizer
    return View("Thanks", guestResponse);
}
}
```

Мы добавили атрибут `HttpGet` для нашего существующего метода действия `RsvpForm`. Это говорит MVC, что данный метод должен использоваться только для GET запросов. Затем мы добавили перегруженную версию `RsvpForm`, который принимает параметр `GuestResponse` и применяет атрибут `HttpPost`. Атрибут говорит MVC, что новый метод будет иметь дело с POST запросами. Обратите внимание, что мы также импортировали пространство имен `PartyInvites.Models` – таким образом мы можем обратиться к типу модели `GuestResponse` без необходимости указывать имя класса. Мы расскажем, как работают наши дополнения в листинге, в следующих разделах.

## **Использование связывания данных модели**

Первый вариант перегруженного метода действия `RsvpForm` обрабатывает то же представление, что и раньше. Она генерирует форму, показанную на [рисунке 2-16](#). Второй вариант перегруженного метода является более интересным из-за параметра, но, учитывая, что метод действия будет вызываться в ответ на HTTP POST запрос, и что `GuestResponse` является типом класса C #, то как они объединены?

Ответом является *связывание данных модели* – чрезвычайно полезная функциональная особенность MVC, когда входные данные разбиваются (парсятся) и пары ключ/значение в HTTP-запросе используются для заполнения свойств типа доменной модели. Этот процесс является противоположностью использования вспомогательных методов HTML; это когда при создании данных формы для отправки клиенту, мы генерировали HTML элементы `input`, где значения атрибутов `id` и `name` были получены из названий свойств классов моделей.

В отличие от этого, со связыванием данных модели, имена элементов `input` используются для указания значений свойств в экземпляре класса модели, которые затем передаются нашему методу действия с `POST` поддержкой.

Модель представления данных является мощной и настраиваемой функцией, которая избавляет от рутины работы напрямую с HTTP-запросами и позволяет нам работать с C# объектами, а не иметь дело со значениями `Request.Form[]` и `Request.QueryString[]`. Объект `GuestResponse`, который передается в качестве параметра нашему методу действия, автоматически заполняется данными из полей формы. Мы рассмотрим подробно модель представления данных, а также в том числе, как ее можно настроить, в главе 22.

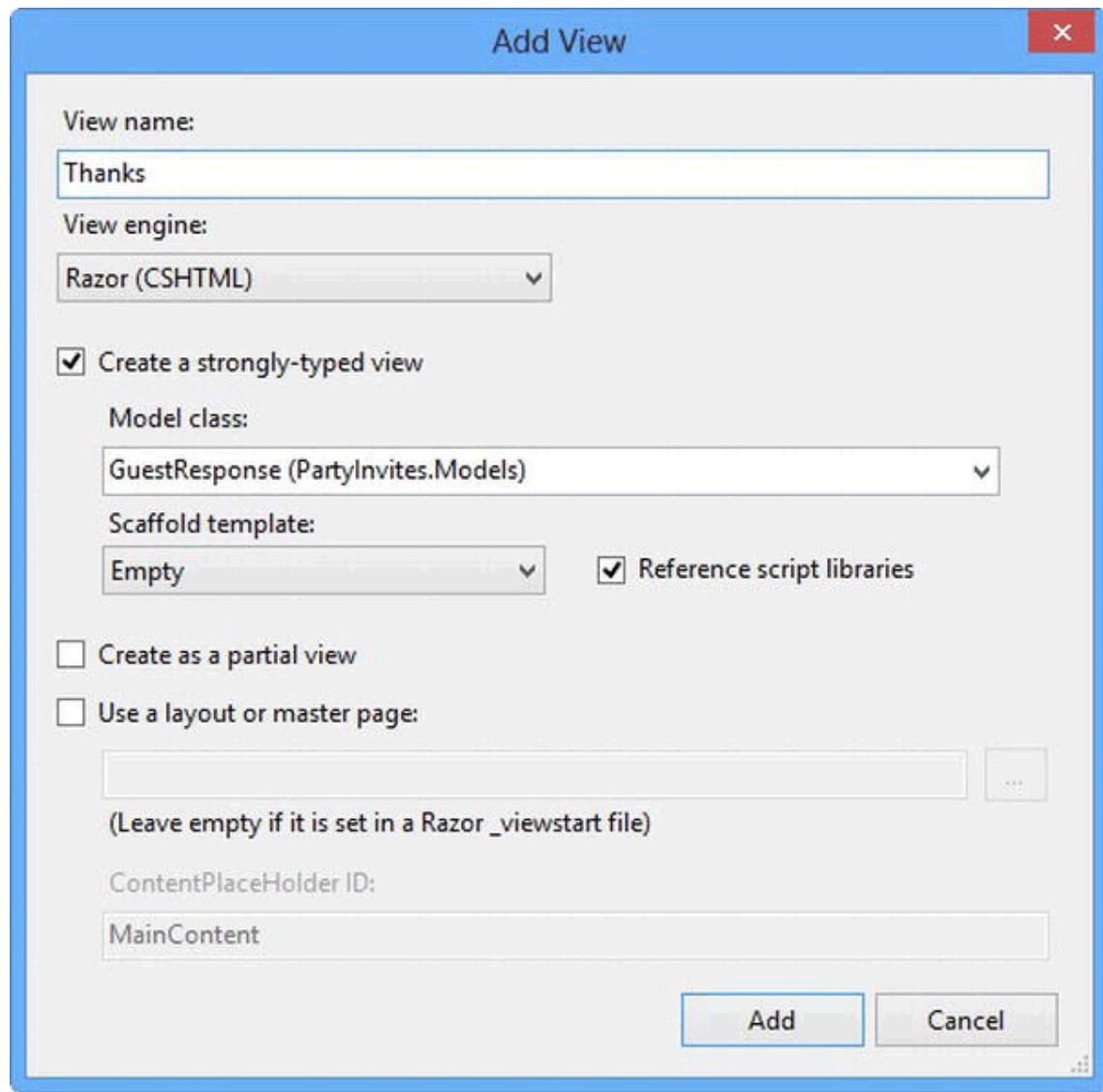
## **Обработка других представлений**

Второй вариант перегруженного метода действий `RsvpForm` также показывает, как мы можем указать MVC обрабатывать конкретное представление, а не представление по умолчанию, в ответ на запрос. Вот соответствующее выражение:

```
return View("Thanks", guestResponse);
```

Этот вызов метода `View` говорит MVC найти и обработать представление, которое называется `Thanks`, и передать представлению наш объект `GuestResponse`. Чтобы создать представление, которое мы указали, щелкните правой кнопкой мыши внутри одного из методов `HomeController` и выберите пункт `Add View` из всплывающего меню. Установите имя представления на `Thanks`, как показано на [рисунке 2-17](#).

**Рисунок 2-17:** Добавление представления `Thanks`



Мы собираемся создать еще одно строго типизированное представление, поэтому поставьте галочку на этом пункте в диалоговом окне `Add View`. Класс данных, который мы выберем для этого представления, должен соответствовать классу, который мы передали представлению при помощи метода `View`. Поэтому убедитесь, что в выпадающем меню выбран `GuestResponse`. Убедитесь, что нет галочки на `Use a layout or master page`, `View engine` установлен на `Razor`, а также `Scaffold template` установлен на `Empty`.

Нажмите кнопку Add, чтобы создать новое представление. Поскольку представление связано с контроллером Home, MVC создаст представление как ~/Views/Home/Thanks.cshtml. Измените новое представление так, чтобы оно соответствовало [листингу 2-15](#): мы выделили то, что нужно добавить.

### Листинг 2-15: Представление Thanks

```
@model PartyInvites.Models.GuestResponse

{@{
    Layout = null;
}

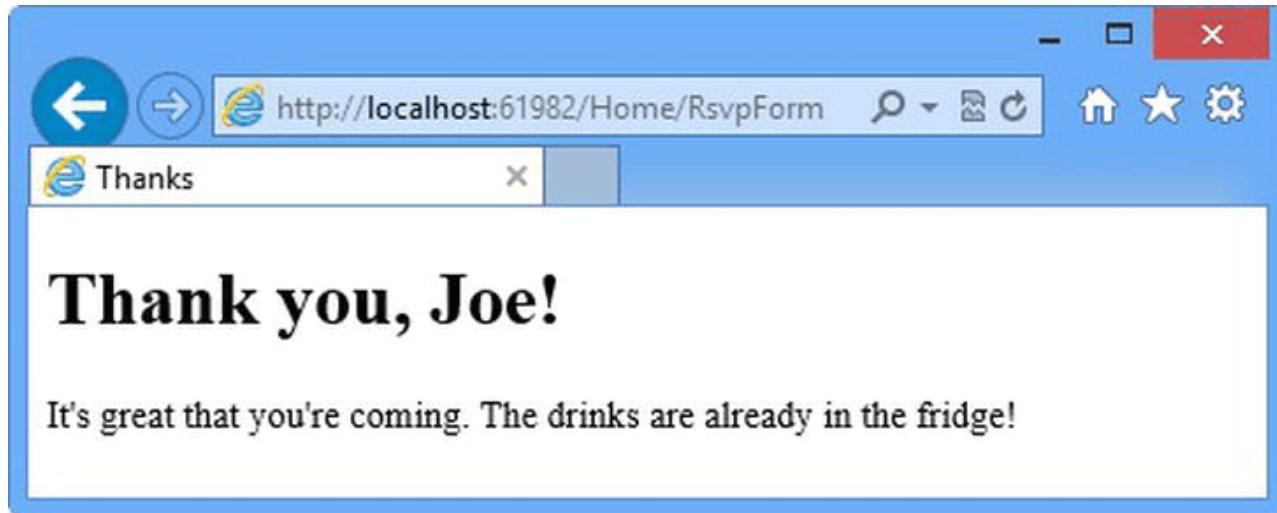
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Thanks</title>
</head>
<body>
    <div>
        <h1>Thank you, @Model.Name!</h1>
        @if (Model.WillAttend == true)
        {
            @:It's great that you're coming. The drinks are already in the fridge!
        }
        else
        {
            @:Sorry to hear that you can't make it, but thanks for letting us know.
        }
    </div>
</body>
</html>
```

Представление Thanks использует Razor для отображения контента на основе значения свойства GuestResponse, которые мы передали методу View в методе действия RsvpForm. Razor оператор @model определяет тип доменной модели, с которым связано представление. Чтобы получить доступ к значению свойства доменного объекта, мы используем Model.PropertyName. Например, чтобы получить значение свойства Name, мы вызываем Model.Name. Не волнуйтесь, если вам не понятен синтаксис Razor, мы объясним это подробно в главе 5.

Теперь, когда мы создали представление Thanks, у нас есть базовый рабочий пример обработки формы при помощи MVC.

Запустите приложение в Visual Studio, нажмите на ссылку RSVP Now, добавьте в форму данные и нажмите на кнопку Submit RSVP. Вы увидите результат, показанный на [рисунке 2-18](#) (хотя он может отличаться, если ваше имя не Джо, и вы сказали, что не сможете присутствовать).

### Рисунок 2-18: Обработанное представление Thanks



## Добавление валидации

Сейчас мы подошли к тому, чтобы добавить валидацию в наше приложение. Если бы мы этого не сделаем, наши пользователи смогут ввести бессмысленные данные или даже отправить пустую форму.

В приложении MVC проверка, как правило, применяется к доменной модели, а не к пользовательскому интерфейсу. Это обозначает, что мы определяем наши критерии валидации в одном месте, и они вступают в силу в любом месте используемого класса модели. ASP.NET MVC поддерживает правила проверки, определяемые атрибутами пространства имен `System.ComponentModel.DataAnnotations`. В [листинге 2-16](#) показано, как эти атрибуты могут быть применены к классу модели `GuestResponse`.

**Листинг 2-16:** Применение валидации к классу модели `GuestResponse`

```
using System.ComponentModel.DataAnnotations;

namespace PartyInvites.Models
{
    public class GuestResponse
    {
        [Required(ErrorMessage = "Please enter your name")]
        public string Name { get; set; }

        [Required(ErrorMessage = "Please enter your email address")]
        [RegularExpression(@"^[\w\.-]+@([\w\.-]+\.)+\w{2,}$", ErrorMessage = "Please enter a valid email address")]
        public string Email { get; set; }

        [Required(ErrorMessage = "Please enter your phone number")]
        public string Phone { get; set; }

        [Required(ErrorMessage = "Please specify whether you'll attend")]
        public bool? WillAttend { get; set; }
    }
}
```

Правила валидации выделены жирным шрифтом. MVC автоматически обнаруживает атрибуты и использует их для проверки данных во время представления модели данных. Обратите внимание, что мы импортировали пространство имен, содержащее правила валидации, поэтому мы можем ссылаться на них без необходимости указывать их имена.

### *Совет*

*Как отмечалось ранее, мы использовали тип `bool?` для свойства `WillAttend`, поэтому мы смогли применить атрибут валидации `Required`. Если бы мы использовали обычный `bool`, значение, которое мы могли бы получить благодаря модели представления данных, могло бы быть только `true` или `false`, и мы не были бы сказать, выбрал ли пользователь значение. Тип `bool?` имеет три возможных значения: `true`, `false` и `null`. Значение `null` будет использовано, если пользователь не выбрал значение, и это заставляет атрибут `Required` сообщить об ошибке валидации.*

---

Мы можем проверить, была ли ошибка валидации проверкой с помощью свойства `ModelState.IsValid` в нашем классе контроллера. В [листе 2-17](#) показано, как это можно сделать в нашем методе действия `RsvpForm` с поддержкой POST.

#### **Листинг 2-17:** Проверка на наличие ошибок при валидации формы

```
[HttpPost]
public ViewResult RsvpForm(GuestResponse guestResponse)
{
    if (ModelState.IsValid)
    {
        // TODO: Email response to the party organizer
        return View("Thanks", guestResponse);
    }
    else
    {
        // there is a validation error
        return View();
    }
}
```

Если ошибки валидации нет, мы говорим MVC обрабатывать представление `Thanks`, как мы делали ранее. Если ошибка валидации есть, мы вновь обрабатываем представление `RsvpForm`, вызвав метод `View` без параметров.

Просто отображать форму, когда есть ошибка, не очень полезно, мы должны дать пользователю некоторую информацию о том, в чем заключается проблема, и почему мы не можем принять его форму. Мы делаем это с помощью вспомогательного метода `Html.ValidationSummary` в представлении `RsvpForm`, как показано в [листе 2-18](#).

#### **Листинг 2-18:** Использование вспомогательного метода `Html.ValidationSummary`

```
@model PartyInvites.Models.GuestResponse

@{
    Layout = null;
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>RsvpForm</title>
</head>
<body>
    @using (Html.BeginForm())
    {
```

```

@Html.ValidationSummary()


Your name: @Html.TextBoxFor(x => x.Name) </p>


Your email: @Html.TextBoxFor(x => x.Email)</p>

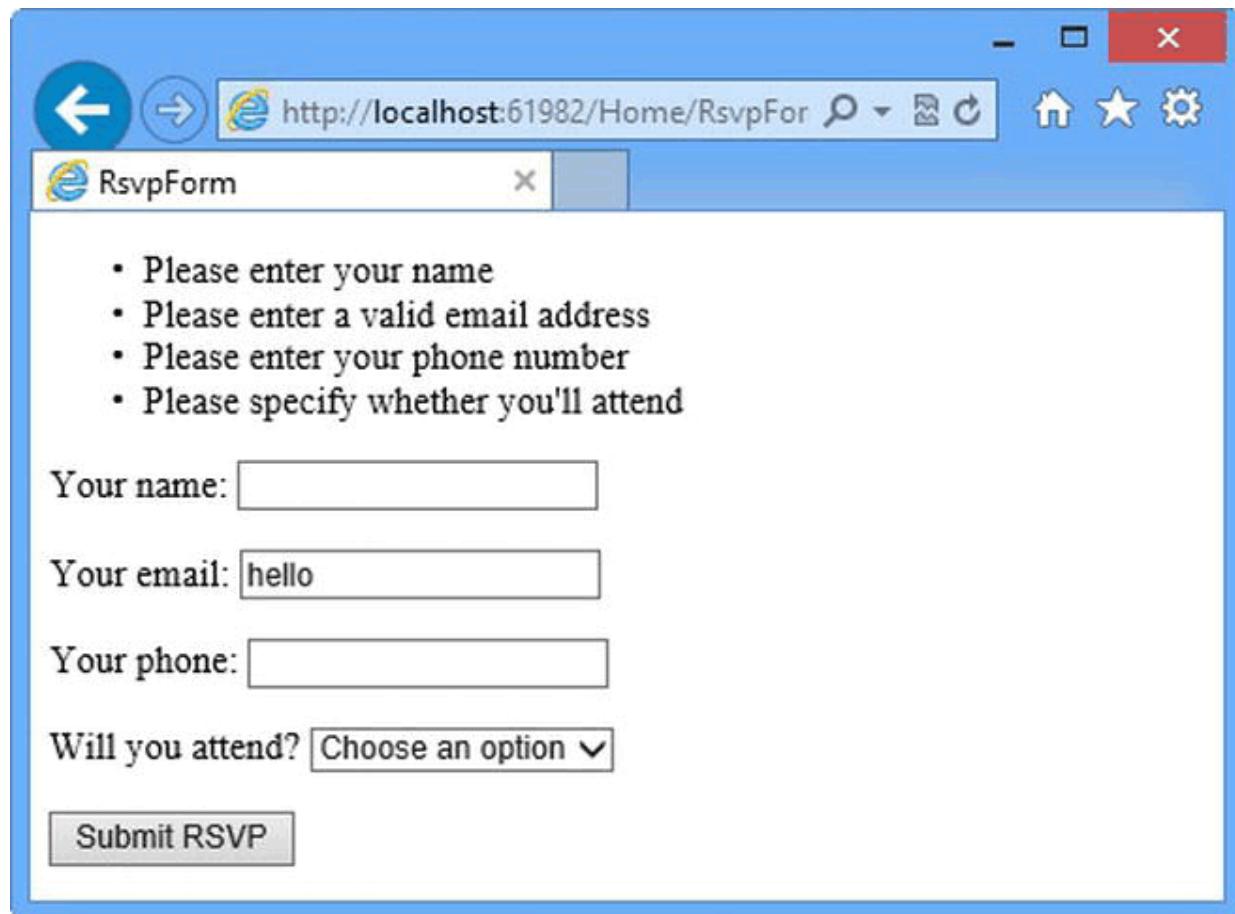

Your phone: @Html.TextBoxFor(x => x.Phone)</p>


Will you attend?
    @Html.DropDownListFor(x => x.WillAttend, new[] {
        new SelectListItem() {Text = "Yes, I'll be there", Value =
bool.TrueString},
        new SelectListItem() {Text = "No, I can't come", Value = bool.FalseString}
    }, "Choose an option")
</p>
<input type="submit" value="Submit RSVP" />
}
</body>
</html>


```

Если ошибок нет, метод `Html.ValidationSummary` создает скрытый элемент списка в качестве заполнителя в форме. MVC делает метку-заполнитель видимой и добавляет сообщения об ошибке, определяемые атрибутами валидации. Вы можете увидеть, как это выглядит, на [рисунке 2-19](#).

**Рисунок 2-19:** Сводка результатов валидации



Пользователю не будет показано представление `Thanks`, пока не будут удовлетворены все правила валидации, которые мы применили к классу `GuestResponse`. Обратите внимание, что данные, введенные в форму, были сохранены и отображаются снова, когда представление показано со сводкой валидации. Это еще одно преимущество связывания данных.

#### Примечание

*Если вы работали с ASP.NET Web Forms, вы знаете, что в Web Forms существует понятие «серверные элементы управления», которые сохраняют состояние, сериализуя значения в скрытом поле формы \_\_VIEWSTATE. Связывание данных ASP.NET MVC не привязано к Web Forms концепции серверных элементов управления, обратной передачи данных или View State. ASP.NET MVC не вводит скрытое поле \_\_VIEWSTATE в обрабатываемые HTML страницы.*

---

## Выделение невалидных полей

Вспомогательные методы HTML, которые создают текстовые поля, выпадающие списки и другие элементы имеют очень удобную функцию, которая может быть использована в сочетании со связыванием данных. Тот же самый механизм, который сохраняет данные, введенные пользователем в форму, также может быть использован для выделения отдельных полей, которые не прошли валидацию.

Если свойство класса модели не прошло валидацию, вспомогательные методы HTML будут генерировать немного другой HTML. В качестве примера, вот HTML, который генерирует вызов `Html.TextBoxFor (x => x.Name)`, когда нет ошибки валидации:

```
<input data-val="true" data-val-required="Please enter your name" id="Name"
name="Name"
type="text" value="" />
```

А вот HTML, который генерирует тот же вызов, когда пользователь не предоставил значение (что является ошибкой валидации, потому что мы применили атрибут Required свойства Name в классе модели `GuestResponse`):

```
<input class="input-validation-error" data-val="true" data-val-required="Please enter
your
name" id="Name" name="Name" type="text" value="" />
```

Мы выделили различия. Это вспомогательный метод добавил класс с именем `input-validation-error`. Мы можем воспользоваться этой функцией, создав таблицу стилей CSS, которая содержит стили для этого класса и других, что применяют различные вспомогательные методы HTML.

Соглашение в MVC проектах заключается в том, что статический контент, такой как таблицы стилей CSS, помещается в папку под названием Content. Мы создали папку Content, нажав правой кнопкой мыши по проекту PartyInvites в Solution Explorer и выбрав из всплывающего меню Add New Folder. Мы создали таблицу стилей, щелкнув правой кнопкой мыши по папке Content, выбрав Add New Item и затем выбрав Style Sheet в диалоговом окне Add New Item. Мы назвали нашу таблицу стилей Site.css, и это имя, которое Visual Studio использует при создании проекта с использованием иного шаблона MVC, а не Empty. Вы можете посмотреть содержимое файла Content/Site.css в [листинге 2-19](#).

### Листинг 2-19: Содержимое файла Content/Site.css

```
.field-validation-error {color: #f00;}
.field-validation-valid { display: none; }
.input-validation-error { border: 1px solid #f00; background-color: #fee; }
.validation-summary-errors { font-weight: bold; color: #f00; }
.validation-summary-valid { display: none; }
```

Чтобы использовать эту таблицу стилей, мы добавили новую ссылку в раздел head представления RsvpForm, как показано в [листе 2-20](#). Вы добавляете представлениям элементы link так же, как в обычном статическом HTML файле.

**Листинг 2-20:** Добавление элемента link в представление RsvpForm

```
@model PartyInvites.Models.GuestResponse

@{
    Layout = null;
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link rel="stylesheet" type="text/css" href("~/Content/Site.css" />
    <title>RsvpForm</title>
</head>
<body>
    @using (Html.BeginForm())
    {
        @Html.ValidationSummary()
        <p>Your name: @Html.TextBoxFor(x => x.Name) </p>
        <p>Your email: @Html.TextBoxFor(x => x.Email)</p>
        <p>Your phone: @Html.TextBoxFor(x => x.Phone)</p>
        <p>
            Will you attend?
            @Html.DropDownListFor(x => x.WillAttend, new[] {
                new SelectListItem() {Text = "Yes, I'll be there", Value =
bool.TrueString},
                new SelectListItem() {Text = "No, I can't come", Value = bool.FalseString}
            }, "Choose an option")
        </p>
        <input type="submit" value="Submit RSVP" />
    }
</body>
</html>
```

*Совет*

*Если вы использовали MVC 3, то могли ожидать, чтобы мы добавим CSS файл к представлению, указав атрибут href как @Href("~/Content/Site.css") или @Url.Content("~/Content/Site.css"). С MVC 4 Razor автоматически обнаруживает атрибуты, которые начинаются с ~/ , и автоматически вставляет для вас @Href или @Url.*

Теперь будет отображаться визуально более очевидная ошибка валидации, если были представлены данные, которые вызвали эту ошибку, как показано на [рисунке 2-20](#).

**Рисунок 2-20:** Автоматически выделенные ошибки валидации

Please enter a valid email address  
Please enter your phone number

Your name: Joe

Your email: hello

Your phone:

Will you attend? Yes, I'll be there

Submit RSVP

## Завершаем пример

Последнее требование к нашему примеру приложения заключается в том, чтобы отправить имейл с завершенными RSVP нашему другу, организатору вечеринки. Мы могли бы сделать это, добавив метод действия, чтобы создать и отправить по электронной почте сообщение, используя e-mail классы .NET Framework. Вместо этого мы собираемся использовать вспомогательный метод WebMail. Это не входит в рамки MVC, но это позволит нам завершить данный пример, не увязнув в деталях создания других средств отправки электронной почты.

### Примечание

*Мы использовали вспомогательный метод WebMail, потому что он позволяет нам с минимальными усилиями продемонстрировать отправку сообщений электронной почты. Однако, в обычной ситуации мы предпочтли бы переложить эту функцию на метод действия. Мы объясним, почему, когда будем описывать архитектурный MVC паттерн в главе 3.*

Мы хотим, чтобы имейл сообщение было отправлено, когда мы обрабатываем представление Thanks. В [листинге 2-21](#) показаны изменения, которые мы должны сделать.

### Листинг 2-21: Использование вспомогательного метода WebMail

```
@model PartyInvites.Models.GuestResponse

@{
    Layout = null;
}

<!DOCTYPE html>
<html>
```

```

<head>
    <meta name="viewport" content="width=device-width" />
    <title>Thanks</title>
</head>
<body>
    @{
        try
        {
            WebMail.SmtpServer = "smtp.example.com";
            WebMail.SmtpPort = 587;
            WebMail.EnableSsl = true;
            WebMail.UserName = "mySmtpUsername";
            WebMail.Password = "mySmtpPassword";
            WebMail.From = "rsvps@example.com";
            WebMail.Send("party-host@example.com", "RSVP Notification",
                Model.Name + " is " + (Model.WillAttend ?? false) ? "" : "not")
                + "attending");
        }
        catch (Exception)
        {
            @:<b>Sorry - we couldn't send the email to confirm your RSVP.</b>
        }
    }
<div>
    <h1>Thank you, @Model.Name!</h1>
    @if (Model.WillAttend == true)
    {
        @:It's great that you're coming. The drinks are already in the fridge!
    }
    else
    {
        @:Sorry to hear that you can't make it, but thanks for letting us know.
    }
</div>
</body>
</html>

```

Мы добавили Razor выражение, которое использует вспомогательный метод `WebMail` для настройки информации о нашем сервере электронной почты, включая имя сервера, требует ли сервер SSL соединения и данных учетной записи. После того как мы все настроили, мы используем метод `WebMail.Send` для отправки электронной почты.

Мы включили весь имейл код в блок `try...catch`, чтобы мы могли предупредить пользователя, если электронная почта не отправилась. Мы делаем это путем добавления текстового блока к выходным данным представления `Thanks`. Лучше было бы отобразить отдельное представление ошибки, если сообщение электронной почты не может быть отправлено, но мы хотели не усложнять наше первое MVC приложение.

## Резюме

В этой главе мы создали новый MVC проект и использовали его, чтобы построить простое MVC приложение по вводу данных, давая вам первое представление об архитектуре MVC Framework и его подходах. Мы пропустили некоторые ключевые особенности (в том числе синтаксис Razor, маршрутизацию (роутинг) и автоматизированное тестирование), но мы вернемся к этим темам в следующих главах.

В следующей главе мы рассмотрим архитектуру MVC, паттерны и технические приемы, которые мы будем использовать на протяжении всей книги.

# MVC паттерн

В главе 7 мы собираемся начать строить более сложный ASP.NET MVC пример. Прежде чем мы начнем копаться в деталях ASP.NET MVC Framework, мы хотим убедиться, что вы знакомы с MVC паттерном и имеете соответствующую философию программирования. В этой главе мы рассмотрим следующее:

- Архитектурный MVC паттерн
- Доменные модели и хранилища
- Создание слабо связанных систем с помощью внедрения зависимости (DI, Dependency injection)
- Основы автоматизированного тестирования

Возможно, вы уже знакомы с некоторыми из идей и соглашений, которые мы обсудим в этой главе, особенно, если вы разрабатывали на ASP.NET и C#. Если же нет, то мы советуем вам внимательно прочитать эту главу: хорошее понимание того, что лежит в основе MVC, может помочь вам рассмотреть все возможности фреймворка, которые мы покажем в данной книге.

## История MVC

Термин *model-view-controller* использовался с конца 1970-х и возник из проекта Smalltalk в Xerox PARC, где он был задуман как способ организации некоторых ранних GUI приложений.

Некоторые мелкие функции исходного MVC паттерна были привязаны к конкретным понятиям Smalltalk, таким как экраны и инструменты (*screens* и *tools*), но более широкие понятия по-прежнему применимы к приложениям, и они особенно хорошо подходят для веб приложений.

Взаимодействие с MVC приложением следует естественному циклу действий пользователя и обновлений представлений, где представление считается не сохраняющим состояние (*stateless*). Это хорошо согласуется с HTTP запросами и ответами, лежащими в основе веб приложений.

Кроме того, MVC заставляет *разделять понятия*: доменная модель и логика контроллера отделены от пользовательского интерфейса. В веб приложениях это обозначает, что HTML хранится отдельно от остальной части приложения, что делает техническую поддержку и тестирование проще и легче. Ruby on Rails привел к возобновлению интереса к MVC, и он остается образцовым «ребенком» MVC. С тех пор появилось много других MVC фреймворков, которые продемонстрировали преимущества MVC, в том числе, конечно, ASP.NET MVC.

## Понимание MVC паттерна

По большому счету MVC паттерн обозначает, что MVC приложение будет разделено как минимум на три части:

- *Модели*, которые содержат или представляют данные, с которыми работают пользователи. Это могут быть простые *модели представления*, которые только представляют данные, передаваемые от контроллера представлению, или они могут быть *доменными моделями*, которые содержат данные домена, а также операции, преобразования и правила работы с этими данными.
- *Представления*, которые используются для того, чтобы обработать некоторые части модели в качестве пользовательского интерфейса.

- *Контроллеры*, которые обрабатывает входящие запросы, выполняют операции для модели и выбирают представления для показа пользователю.

Модели являются определением вселенной, в которой работает ваше приложение. В банковском приложении, например, модель представляет собой все в банке, что поддерживает приложение: то есть счета, главную книгу и кредитные лимиты для клиентов, а также операции, которые могут быть использованы для манипулирования данными в модели, такие как депонирование средств и осуществление изъятия со счетов. Модель также несет ответственность за сохранение общего состояния и согласованности данных, например, она гарантирует, что все сделки будут добавлены в книгу, и что клиент не сможет снять больше денег, чем он может, или больше денег, чем имеет банк.

Модели также определяются тем, за что они *не* несут ответственность: модели не занимаются UI и обработкой запросов – это обязанности *представлений и контроллеров*. *Представления* содержат логику, необходимую для отображения элементов модели пользователю, и больше ничего. Они не имеют прямого понимания модели и никоим образом напрямую не сообщаются с моделью.

*Контроллеры* являются мостом между представлениями и моделью: запросы приходят от клиента и обслуживаются контроллером, который выбирает соответствующее представление для показа пользователю и, при необходимости, соответствующие действия, которые нужно выполнить с моделью.

Каждый элемент MVC архитектуры является четко определенным и автономным – это называется *разделением понятий*. Логика, которая манипулирует данными в модели, содержится *только* в модели; логика, которая отображает данные, находится *только* в представлении, а код, который обрабатывает запросы пользователей и вводные данные, содержится *только* в контроллере. С четким разделением обязанностей между каждой из частей ваше приложение будет легче поддерживать и расширять, независимо от того, насколько большим оно будет становиться.

## Понимание доменной модели

Наиболее важной частью MVC приложения является *доменная модель*. Мы создаем модели путем выявления реальных лиц, операций и правил, которые существуют в отрасли или деятельности. Они должны поддерживаться нашим приложением, и они известны как *домен*.

Затем мы создаем программное представление домена – *доменную модель*. Для наших целей доменная модель представляет собой набор типов C# (классы, структуры и т.д.), известных под общим названием *доменные типы*. Операции из домена представлены методами, определенными в доменных типах, а доменные правила выражаются в логике внутри этих методов, или, как мы видели в предыдущей главе, в применении C# атрибутов к методам. Когда мы создаем экземпляра доменного типа, чтобы представить определенный фрагмент данных, мы создаем *доменный объект*. Доменные модели, как правило, постоянны – есть много разных способов достижения этого, но реляционные базы данных остаются наиболее распространенным выбором.

Короче говоря, доменная модель является отдельным авторитетным определением бизнес данных и процессов внутри вашего приложения. *Постоянная* доменная модель также является вспомогательным определением состояния доменного представления.

### *Совет*

*Распространенный способ обеспечения разделения доменной модели и остальной части ASP.NET MVC приложения заключается в том, чтобы поместить модель в отдельную сборку C#. Таким образом, вы можете создать ссылки на доменную*

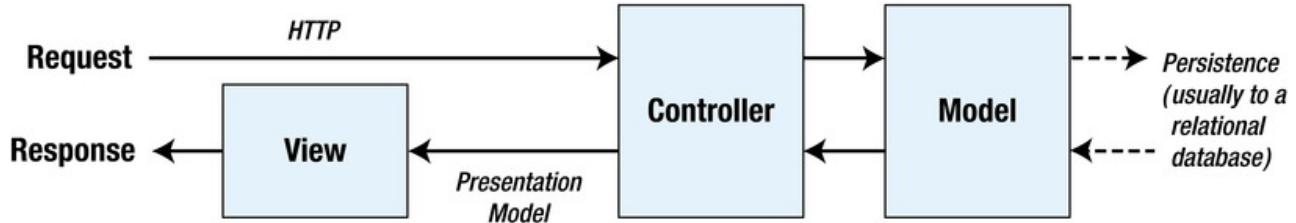
*модели из других частей приложения, но нужно убедиться, что в обратном направлении ссылок нет. Это особенно полезно в крупномасштабных проектах. Мы будем использовать такой подход в примере, который мы начнем строить в главе 7.*

## ASP.NET реализация MVC

В MVC контроллеры являются C# классами, как правило, производными от класса `System.Web.Mvc.Controller`. Каждый открытый (`public`) метод в классе, производный от `Controller`, называется *методом действия*, который связан с настраиваемым URL через систему маршрутизации (роутингом) ASP.NET. При отправке запроса на URL, связанный с методом действия, исполняются выражения в классе контроллера для того, чтобы выполнить некоторые операции над доменной моделью, а затем выбрать представление для отображения клиенту. На [рисунке 3-1](#) показано взаимодействие между контроллером, моделью и представлением.

**Рисунок 3-1:** Взаимодействие основных компонентов в MVC приложении

ASP.NET MVC Fr



Framework обеспечивает поддержку выбора движков представления. Более ранние версии MVC использовали стандартный ASP.NET движок представления, который обрабатывал ASPX страницы с помощью модернизированной версии синтаксиса разметки Web Forms. MVC 3 ввел движок представления Razor, который был усовершенствован в MVC 4 и который использует полностью другой синтаксис (описанный в главе 5). Visual Studio обеспечивает поддержку IntelliSense для обоих движков представления, что сильно упрощает работу с данными представления, отправленными контроллером.

ASP.NET MVC не применяет никаких ограничений на реализацию вашей доменной модели. Вы можете создать модель с помощью обычных C# объектов и осуществлять хранение при помощи любой из баз данных, объектно-реляционных фреймворков или других инструментов хранения данных, поддерживаемых .NET. Visual Studio создает папку `/Models` в рамках шаблона MVC проекта. Это подходит для простых проектов, но в более сложных приложениях, как правило, доменные модели определяются в отдельный проект Visual Studio. Мы обсудим реализацию доменной модели далее в этой главе.

## Сравнение MVC с другими паттернами

MVC – это не единственный архитектурный паттерн программного обеспечения, конечно, есть много других, и некоторые из них (или по крайней мере были) чрезвычайно популярны. Мы можем многое узнать об MVC, рассмотрев другие паттерны. В следующих разделах мы кратко опишем различные подходы к структурированию приложения и сопоставим их с MVC. Некоторые из паттернов являются близкими вариациями на тему MVC, в то время как другие совершенно отличаются.

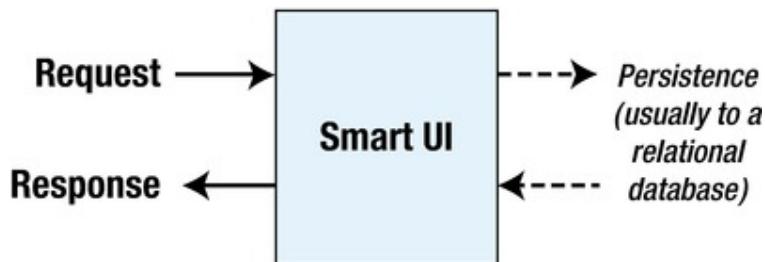
Мы не утверждаем, что MVC является идеальным паттерном для всех ситуаций. Мы оба являемся сторонниками того, чтобы выбрать наилучший подход к решению проблемы. Как вы видите, бывают ситуации, когда мы понимаем, что некоторые конкурирующие паттерны так же полезны или даже лучше, чем MVC. Мы рекомендуем вам делать обоснованный и осознанный выбор при выборе паттерна. Тот факт, что вы читаете эту книгу, показывает, что у вас уже есть определенный интерес к MVC паттерну, но мы думаем, это всегда полезно рассмотреть и более широкую перспективу.

## Паттерн Smart UI

Один из наиболее распространенных паттернов известен как smart UI. Большинство программистов в какой-то момент своей карьеры создавали smart UI приложения, мы, конечно же, тоже. Если вы использовали Windows Forms и ASP.NET Web Forms, то и вы тоже.

Чтобы создать smart UI приложение, разработчики выстраивают пользовательский интерфейс, обычно путем перетаскивания набора компонентов или элементов управления на дизайнерскую поверхность. Элементы управления сообщают о взаимодействии с пользователем, представляя события для нажатия кнопок, нажатия клавиш, движения мыши и так далее. Разработчик добавляет код в ответ на эти события в серии обработчиков событий – небольших блоков кода, которые вызываются, когда происходит определенное событие для определенного компонента. И тут мы заканчиваем с монолитным приложением, как показано на [рисунке 3-2](#) – код, который обрабатывает пользовательский интерфейс и логику смешиваются вместе, понятия вообще не разделены. Код, который определяет допустимые значения для вводимых данных, запросов данных или изменяет учетную запись пользователя, делится на маленькие кусочки, соединенные вместе по порядку, в котором ожидаются события.

Рисунок 3-2: Smart UI паттерн



Самым большим недостатком этой конструкции является то, что ее трудно поддерживать и расширять: смешивание доменной модели и кода бизнес логики с кодом пользовательского интерфейса приводит к дублированию, где тот же фрагмент бизнес логики копируется и вставляется для поддержки вновь добавленного компонента. Поиск всех дублирующих частей и внесение правок может быть трудным, а в сложном smart UI приложении иногда почти невозможно добавить новую функцию, не ломая существующую. Тестирование smart UI приложений также может быть трудным, единственный способ – это имитации взаимодействий с пользователем, которая далека от идеала и не обеспечивает основу для полного тестирования.

В мире MVC Smart UI часто называют *анти-паттерном*, которого следует избегать любой ценой. Эта антипаттерн возникла, по крайней мере, частично потому, что люди пришли к MVC в поисках альтернативы, проведя часть своей карьеры, пытаясь развивать и поддерживать Smart UI приложения. Это, безусловно, относится и к нам, и мы оба несем в себе следы этих долгих лет, но мы не отвергаем smart UI паттерн полностью. Не все гладко в smart UI паттерне, но есть и положительные аспекты в таком подходе. Smart UI приложения быстро и легко разрабатывать: создатели инструментария приложили много усилий для того, чтобы разработка была приятной, и

даже самый неопытный программист может создать нечто профессионально выглядящее и достаточно функциональное в течение нескольких часов.

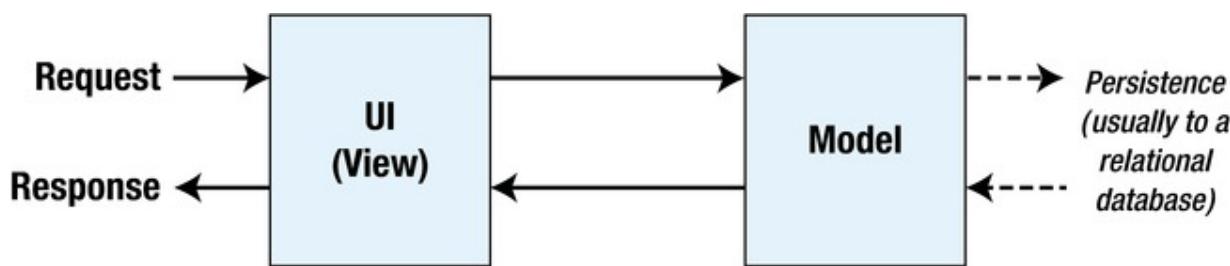
Самая большая слабая сторона Smart UI приложений – сопровождение и поддержка – не видна при разработке небольших приложений. Если вы создаете простой инструмент для небольшой аудитории, то Smart UI приложение может быть идеальным решением. Дополнительная сложность MVC приложений просто не оправдана.

Наконец, Smart UI идеально подходит для создания прототипов пользовательского интерфейса, эти инструменты конструирования действительно хороши. Если вы сидите с клиентом и хотите понять требования к интерфейсу, инструменты Smart UI могут быть быстрым и отзывчивым способом создания и тестирования различных идей.

### Архитектура Model-View

Область, в которой, как правило, возникают проблемы с поддержкой в Smart UI приложениях – это бизнес-логика, и внесение изменений может стать очень неприятным процессом. Улучшения в этой области предлагает архитектура *model-view* (модель-представление), которая вытаскивает бизнес-логику в отдельную доменную модель. При этом данные, процессы и правила все сосредоточены в одной части приложения, как показано на [рисунке 3-3](#).

Рисунок 3-3: model-view паттерн

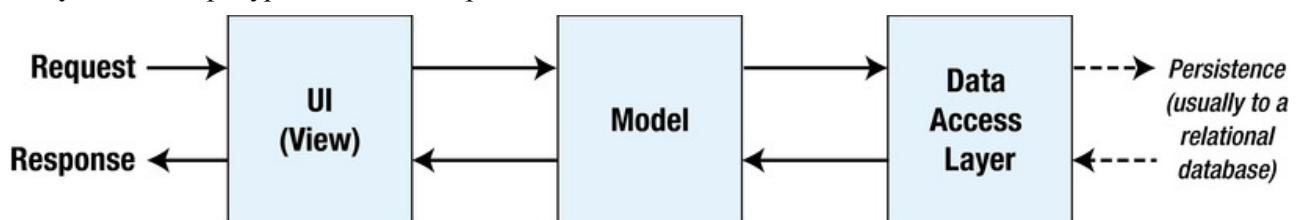


Архитектура model-view является значительным улучшением по сравнению со Smart UI – например, ее гораздо легче поддерживать. Тем не менее, возникают две проблемы. Первое, поскольку пользовательский интерфейс и доменная модель настолько тесно интегрированы, затрудняется выполнение юнит тестирования. Вторая проблема вытекает из практики, а не из определения паттерна. Эта модель обычно содержит массу кода доступа к данным, а это значит, что модель данных содержит не только бизнес данные, операции и правила.

### Классическая трехуровневая (three-tier) архитектура

Для решения проблемы с архитектурой model-view *трехуровневая* или трехслойная (three-tier или three-layer) модель отделяет код от доменной модели и помещает его в новый компонент под названием DAL. Это показано на [рисунке 3-4](#).

Рисунок 3-4: Трехуровневый паттерн



Это большой шаг вперед. Трехуровневая архитектура является наиболее широко используемым паттерном для бизнес-приложений. Она не имеет ограничений относительно того, как реализован пользовательский интерфейс, и обеспечивает хорошее разделение понятий, не будучи слишком сложной. Также DAL может быть создан таким образом, что юнит тестирование будет относительно легким. Вы можете увидеть очевидное сходство между классическим трехуровневым приложением и MVC паттерном. Разница состоит в том, что поскольку UI слой напрямую соединен с «click-and-event» GUI фреймворком (таким как Windows Forms и ASP.NET Web Forms), становится почти невозможным выполнение автоматизированных юнит тестов. И поскольку UI часть трехуровневого приложения может быть очень сложной, мы имеем много кода, который не может быть тщательно протестирован.

В худшем случае, отсутствие в трехуровневой модели управления над слоем UI означает, что многие такие приложения в конечном итоге выглядят как плохо замаскированные Smart UI приложения, без реального разделения понятий. И в итоге мы получаем нетестируемое, не поддерживаемое приложение, которое вдобавок является очень сложным.

## Вариации MVC

Мы уже изучили основные принципы создания MVC приложений, и особенно то, как они относятся к реализации ASP.NET MVC. Другие интерпретируют аспекты паттерна по-другому и корректируют или иным образом адаптируют MVC для своих проектов. В следующих разделах мы представим краткий обзор двух наиболее распространенных вариантов на тему MVC. Понимание этих вариантов не является необходимым для работы с ASP.NET MVC. Мы включили эту информацию для полноты.

### Model-View-Presenter паттерн

MVP является вариацией MVC, который разработан, чтобы соответствовать GUI платформам, сохраняющим состояние (stateful), таким как Windows Forms или ASP.NET Web Forms, и это стоящая попытка представить лучшие аспекты Smart UI без тех проблем, которые он обычноносит.

В этом паттерне презентер (presenter) имеет те же обязанности, что и MVC контроллер, но он также имеет более непосредственное отношение к представлениям, напрямую управляя значениями, отображаемыми в UI компонентах, в зависимости от действий и данных, введенных пользователем. Существуют две реализации этого паттерна:

- Реализация *пассивного представления*, в котором представление не содержит логики – это контейнер для элементов управления UI, которые напрямую управляются презентером.
- Реализация *надзирающего контроллера*, в котором представление может быть ответственным за некоторые элементы логики представления, такие как связанные данные, и к источнику данных можно ссылаться из доменной модели.

Разница между этими двумя подходами касается того, насколько умным является представление. В любом случае, презентер отделяется от GUI фреймворка, что делает логику презентера простой и подходит для модульного тестирования.

### Model-View-View Model паттерн

MVVM паттерн является самой последней вариацией MVC. Он возник в 2005 году в команде Microsoft, которая разрабатывала технологию, ставшую Windows Presentation Foundation (WPF) и Silverlight.

В MVVM паттерне модели и представления играют те же роли, что и в MVC. Разница состоит в MVVM концепции модели представления, которая является абстрактным представлением пользовательского интерфейса. Как правило, это C# класс, который раскрывает оба свойства для данных, которые будут отображаться в интерфейсе, и операции по данным, которые могут быть вызваны из пользовательского интерфейса. В отличие от MVC контроллера, модель представления MVVM не имеет понятия, что существует представление (или любая конкретная UI технология). MVVM представление использует *связывающую* функцию WPF/Silverlight для двунаправленного связывания свойств, предоставляемых элементами управления в представлении (пункты в выпадающем меню или результат нажатия кнопки), со свойствами, предлагаемыми моделью представления.

MVVM тесно сотрудничает со связывающими функциями WPF и поэтому это не тот паттерн, который легко применить и к другим платформам.

#### *Совет*

*MVC также использует термин модель представления, но он относится к простому классу модели, который используются только для передачи данных из контроллера в представление. Мы должны четко различать модель представления и доменную модель, которая является сложным представлением данных, операций и правил.*

---

## Проблемно-ориентированное программирование (DDD)

Мы уже описали, как доменная модель представляет реальный мир в вашем приложении, она содержит представления ваших объектов, процессов и правил. Доменная модель является сердцем MVC приложения, а все остальное, в том числе представления и контроллеры – это всего лишь средства для взаимодействия с доменной моделью.

ASP.NET MVC не диктует того, какая технология должна использоваться для доменной модели. Вы вольны выбрать любую технологию, которая будет взаимодействовать с .NET Framework, и тут есть много вариантов. Тем не менее, ASP.NET MVC предлагает инфраструктуру и соглашения, чтобы помочь подключить классы доменной модели к контроллерам и представлениям, а также к самому MVC Framework. Есть три ключевые функциональные возможности:

- *Связывание данных модели* является функцией, которая автоматически заполняет объекты модели, используя входные данные, как правило, отправленные из HTML формы.
- *Метаданные модели* позволяют описать фреймворку смысл классов модели. Например, вы можете предоставить читабельное описание их свойств или дать подсказки о том, как они должны отображаться. MVC Framework может автоматически представить изображение или редактор UI для классов модели в представлениях.
- *Валидация*, которая выполняется во время связывания данных и применяет правила, которые могут быть определены как метаданные.

Мы кратко коснулись связывания данных и валидации, когда мы строили наше первое MVC приложение в [главе 2](#), и мы вернемся к этим темам и дальнейшему их изучению в главах 22 и 23. На данный момент, мы собираемся отложить реализацию ASP.NET MVC в сторону и подумать о доменном моделировании как о самостоятельной деятельности. Мы собираемся создать простую

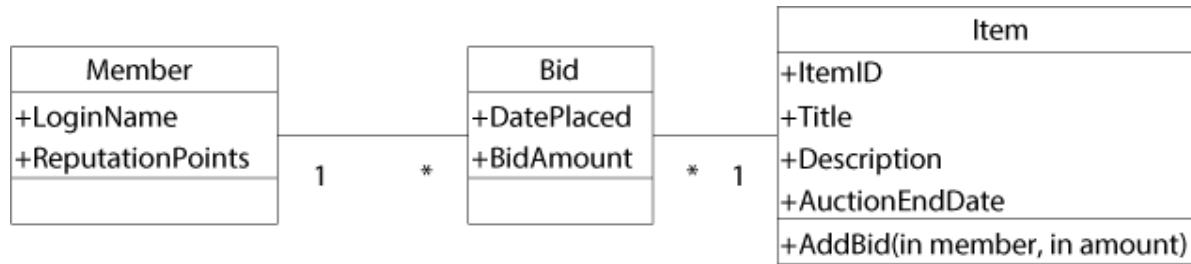
доменную модель с помощью .NET и SQL Server, используя некоторые основные технологические приемы из мира DDD.

## Построение доменной модели

У вас, наверное, уже был опыт мозгового штурма по созданию доменной модели. Обычно сюда включены разработчики, бизнес эксперты, большое количество кофе, печенья и ручки с маркерами. Через некоторое время люди в комнате приходят к начальному общему знаменателю, и тогда возникает первый проект доменной модели. (Мы опускаем рассказ о многочасовых разногласиях. Достаточно сказать, что разработчики будут тратить первые часы, требуя от бизнес экспертов реальных задач, а не взятых из научной фантастики, в то время как бизнес эксперты будут выражать удивление и обеспокоенность, что время и смета расходов аналогичны тем, что NASA требует, чтобы достичь Марса. Кофе имеет важное значение в решении таких противоречий и в таком противостоянии, в конечном итоге, мочевой пузырь у всех будет настолько полным, что все пойдут на компромисс, и наметится прогресс в решении задачи).

Вы могли заканчивать чем-то похожим, что изображено на [рисунке 3-5](#) и что является отправной точкой для данного примера – простая доменная модель для аукционного приложения.

**Рисунок 3-5:** Первый набросок модели для аукционного приложения



Эта модель содержит набор элементов Members, каждый из которых содержит набор элементов Bids. Каждый Bid предназначен для одного Item, а каждый Item может содержать несколько Bid от разных Members.

## Повсеместно используемый язык

Ключевым преимуществом реализации вашей доменной модели в качестве отдельного компонента является то, что вы можете установить по своему выбору язык и терминологию. Вам стоит попытаться найти терминологию для ее объектов, операций и отношений, которые будут понятны не только разработчикам, но и бизнес экспертам. Мы рекомендуем вам адаптировать доменную терминологию, когда модель уже существует. Например, если то, что разработчик будет называть пользователями и ролями (*users* и *roles*), известно в домене как агенты и разрешения (*agents* и *clearances*), то мы рекомендуем вам принять последний вариант в вашей доменной модели.

При моделировании таких понятий, для которых у экспертов нет нужной терминологии, вам нужно прийти к общему соглашению о том, как вы будете их называть, создавая повсеместно используемый язык, который будет проходить через доменную модель.

Разработчики, как правило, говорят на языке кода – именами классов, названиями таблиц баз данных и так далее. Бизнес эксперты не понимают эти термины, да они и не должны. Бизнес эксперт с небольшими техническими знаниями – это опасная вещь, потому что он будет постоянно фильтровать требования через свое понимание того, на что способна технология. И если так случится, вы не получите истинного понимания того, что нужно бизнесу.

Этот подход также помогает избежать в приложении сверхобщения. Программисты, как правило, зачастую хотят смоделировать все возможные реалии бизнеса, а не конкретно то, что требует бизнес. В модели аукциона такое могло бы произойти, если бы мы изменили `members` и `items` общими понятиями `resources` и `relationships`. Когда мы создаем доменную модель, которая не совсем соответствует моделируемому домену, мы упускаем возможность получить любую реальную картину бизнес процесса и, в будущем, представление бизнес процесса может оказаться неправильным. Эти ограничения не являются запретом, они являются маяком, который направляет вашу работу в правильное русло.

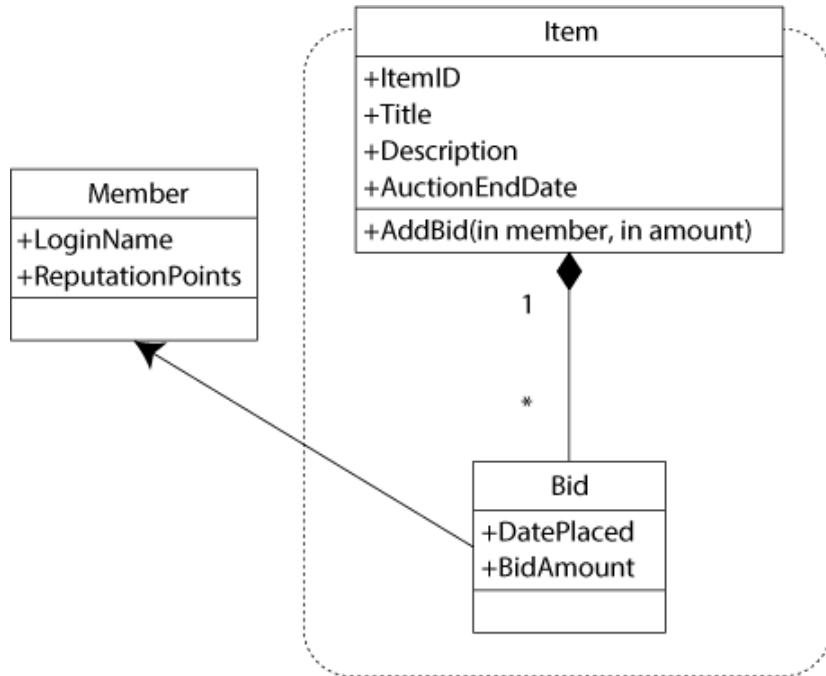
Связь между повсеместно используемым языком и доменной моделью не должна быть поверхностной: DDD эксперты предполагают, что любые изменения в повсеместно используемом языке должны приводить к изменению модели. Если вы позволите модели не быть синхронизированной с доменом, вы создадите промежуточный язык, который будет различаться у модели и домена, и это может в дальнейшем привести к серьезным проблемам. Вы создадите специальный класс людей, которые могут говорить на обоих языках, и тогда они начнут фильтровать требования, основываясь на своем неполном понимании обоих языков.

## Агрегаты и упрощение

На [рисунке 3-5](#) представлена хорошая отправная точка для нашей доменной модели, но тут нет никаких полезных указаний о реализации модели с использованием C# и SQL Server. Если мы загрузим в память элемент `Member`, должны ли мы также загрузить `Bids` и `Items`, связанные с ним? И если это так, нужно ли нам загрузить все другие `Bids` для этих `Items`, а также `Members`, которые связаны с этими `Bids`? Когда мы удаляем объект, должны ли мы удалить также связанные с ним объекты, и если да, то какие? Если мы выберем для реализации хранилище документов вместо реляционной базы данных, какая коллекция объектов будет представлять собой единый документ? Мы этого не знаем, а наша доменная модель не дает нам никаких ответов на эти вопросы.

В данном случае DDD предлагает организовать доменные объекты в группы, называемые агрегатами. На [рисунке 3-6](#) показано, как мы можем объединить объекты в нашей доменной модели аукциона.

**Рисунок 3-6:** Аукционная доменная модель с агрегатами



Суть агрегата заключается в группировке нескольких объектов доменной модели: есть ключевая сущность (*root entity*), которая используется для определения цельного агрегата, и она действует как «босс» для валидации и сохранения операций. Агрегат рассматривается как единое целое с точки зрения изменения данных, поэтому мы должны создать агрегаты, которые представляют собой отношения, имеющие смысл в контексте доменной модели, и создать операции, которые логически соответствуют реальным бизнес процессам: иными словами, мы должны создавать агрегаты, группируя объекты, которые изменяются как группа.

Ключевое правило DDD гласит, что объекты за пределами конкретного экземпляра агрегата могут быть связаны только с root entity, а не с любым другим объектом внутри агрегата (на самом деле, идентичность не корневого объекта должна быть уникальной только в пределах своего агрегата). Это правило подтверждает представление о работе с объектами внутри агрегата как с единым целым.

В нашем примере `Members` и `Items` являются ключевыми сущностями агрегата, в то время как `Bids` могут быть доступны только в контексте элемента `Item`, который является корневым в своем агрегате. Элементы `Bids` могут быть связаны с элементами `Members` (которые являются основными сущностями, root entity), но `Members` не могут непосредственно ссылаться на `Bids` (поскольку они не являются корневыми).

Одно из преимуществ агрегатов состоит в том, что они упрощает систему отношений между объектами в доменной модели, и часто они могут дать дополнительное понимание природы домена, который моделируется. В сущности, создание агрегатов ограничивает отношения между объектами доменной модели, так что они становятся больше похожими на отношения, которые существуют в реальном домене. В [листе 3-1](#) показано, как выглядит наша доменная модель, выраженная в C#.

### Листинг 3-1: Аукционная доменная модель, выраженная в C#

```
public class Member
{
    public string LoginName { get; set; } // Уникальный ключ
    public int ReputationPoints { get; set; }
}

public class Item
{
    public int ItemID { get; private set; } // Уникальный ключ
    public string Title { get; set; }
    public string Description { get; set; }
    public DateTime AuctionEndDate { get; set; }
    public IList<Bid> Bids { get; set; }
}

public class Bid
{
    public Member Member { get; set; }
    public DateTime DatePlaced { get; set; }
    public decimal BidAmount { get; set; }
}
```

Обратите внимание, как мы легко смогли понять односторонний характер отношений между `Bids` и `Members`. Мы также смогли смоделировать некоторые другие ограничения, например, элементы `Bids` остаются неизменными (это представляет собой общую стратегию аукциона, когда ставки не могут быть изменены, как только они сделаны). Применение агрегации позволило нам создать более полезную и точную доменную модель, которую мы смогли с легкостью представить в C#.

В общем, агрегаты добавляют структуру и точность доменной модели. Они упрощают применение валидации (ключевая сущность становится ответственной за валидацию состояния всех объектов в агрегате), и очевидны юниты, которые нужно сохранять. И поскольку агрегаты, по сути, являются атомарными единицами нашей доменной модели, они также являются подходящими единицами для управления транзакциями и каскадного удаления из базы данных.

С другой стороны, они накладывают ограничения, которые иногда могут показаться искусственными, потому что зачастую они и *есть* искусственные. Агрегаты обычно появляются в базах данных документа, но они не являются родным понятием в SQL Server и в большинстве инструментов ORM. Поэтому если вы хотите их хорошо реализовать, вашей команде понадобится дисциплина и эффективный обмен информацией.

## Определение репозиториев

В какой-то момент нам нужно будет обеспечить постоянство нашей доменной модели: это, как правило, осуществляется с помощью реляционных, объектных или документальных баз данных. Постоянство не является частью нашей доменной модели – это *самостоятельное* или *ортогональное* понятие в нашем принципе разделения понятий. Это означает, что мы не хотим смешивать код, который обрабатывает постоянство, с кодом, который определяет доменную модель.

Обычным способом обеспечить разделение между доменной моделью и системой постоянства является определение *репозиториев* – это объекты представления основной базы данных (или хранилища файлов, которое вы выбрали). Вместо того чтобы работать непосредственно с базой данных, доменная модель вызывает методы, определенные в репозитории, который в свою очередь делает запросы в базу данных для хранения и извлечения данных модели. Это позволяет изолировать модель от реализации постоянства.

Соглашение заключается в определение отдельной модели данных для каждого агрегата, потому что агрегаты являются естественными единицами для сохранения постоянства. В случае нашего аукциона, например, мы можем создать два репозитория: один для `Members` и один для `Items` ( обратите внимание, что нам не нужен репозиторий для `Bids`, потому что они будут сохранены как часть агрегата `Items`). В [листе 3-2](#) показано, как могут быть определены эти репозитории.

**Листинг 3-2:** C# классы репозиториев для доменных классов `Member` и `Item`

```
public class MembersRepository
{
    public void AddMember(Member member)
    {
        /* Реализуй меня */
    }
    public Member FetchByLoginName(string loginName)
    {
        /* Реализуй меня */
        return null;
    }

    public void SubmitChanges()
    {
        /* Реализуй меня */
    }
}
public class ItemsRepository
{
    public void AddItem(Item item)
    {
```

```

    /* Реализуй меня */
}
public Item FetchByID(int itemID)
{
    /* Реализуй меня */
    return null;
}
public IList<Item> ListItems(int pageSize, int pageIndex)
{
    /* Реализуй меня */
    return null;
}
public void SubmitChanges()
{
    /* Реализуй меня */
}
}

```

Обратите внимание, что репозитории касаются только загрузки и сохранения данных: они не содержат доменной логики вообще. Мы можем завершить классы репозиториев путем добавления выражений для каждого метода, который выполняет операции по сохранению и получению для соответствующего механизма сохранения. В главе 7 мы начнем строить более сложные и реальные MVC приложения, и мы покажем вам, как использовать Entity Framework для реализации ваших репозиториев.

## Строительство слабосвязанных компонентов

Как мы уже говорили, одна из наиболее важных особенностей шаблона MVC заключается в том, что он позволяет разделять понятия. Мы хотим, чтобы компоненты в нашем приложении были настолько независимыми, насколько это возможно, и были так взаимосвязаны, чтобы мы могли ими управлять.

В нашей идеальной ситуации каждый компонент ничего не знает о других компонентах и работает с другими областями приложения через абстрактные интерфейсы. Это известно как слабая связь компонентов, и она упрощает тестирование и возможность изменений нашего приложения.

Простой пример поможет понять все это. Если мы пишем компонент `MyEmailSender`, который будет отправлять электронную почту, мы хотели бы реализовать интерфейс, который определяет все открытые функции, необходимые для отправки электронной почты, который мы бы назвали `IEmailSender`.

Любой другой компонент нашего приложения, которому нужно отправить имейл, скажем, например, вспомогательный метод сброса пароля `PasswordResetHelper`, может отправить электронную почту, только ссылаясь на методы интерфейса. Как показано на [рисунке 3-7](#), между `PasswordResetHelper` и `MyEmailSender` не существует прямой зависимости.

**Рисунок 3-7:** Использование интерфейсов для разделения компонентов



Вводя `IEmailSender`, мы гарантируем, что не существует прямой зависимости между `PasswordResetHelp` и `MyEmailSender`. Мы могли бы заменить `MyEmailSender` другим

компонентом для отправки электронной почты или даже использовать в целях тестирования мок-объекты. Мы вернемся к теме мок-объектов в главе 6.

#### Примечание

*Не все отношения должны быть разделены с помощью интерфейса. Решение об этом основывается на следующих тезисах: насколько сложным является приложение, какой вид тестирования требуется и какова вероятная долгосрочная поддержка. Например, в небольшом и простом ASP.NET MVC приложении мы могли бы не разделять контроллеры от доменной модели.*

## Использование внедрения зависимости (Dependency Injection)

Интерфейсы помогают нам разделять компоненты, но у нас по-прежнему есть проблема: C# не предоставляет встроенного способа для легкого создания объектов, реализующих интерфейсы, за исключением создания интерфейсов конкретного компонента. В конечном итоге, у нас есть код, показанный в [листинге 3-3](#).

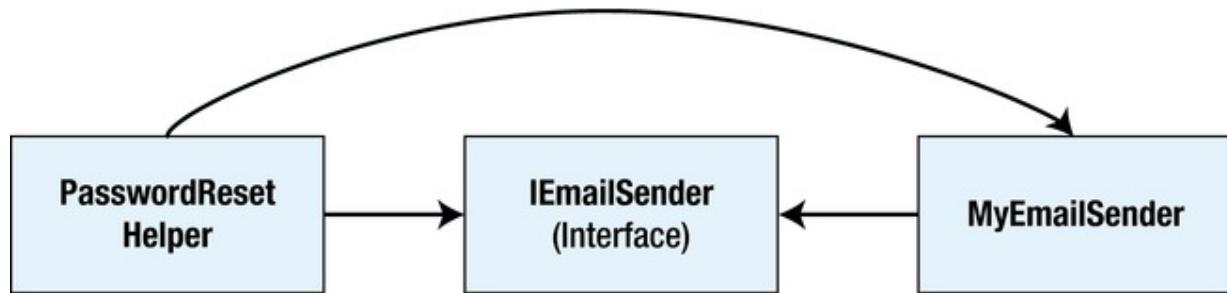
**Листинг 3-3:** Создание экземпляра конкретного класса для осуществления реализации интерфейса

```
public class PasswordResetHelper {  
    public void ResetPassword() {  
        IEmailSender mySender = new MyEmailSender();  
        //...вызов методов интерфейса для конфигурации информации по имейлу  
        mySender.SendEmail(); } }
```

Мы всего лишь частично на пути к слабосвязанным компонентам: класс PasswordResetHelper настраивает и отправляет электронную почту через интерфейс IEmailSender, но для создания объекта, который реализует этот интерфейс, он должен был создать экземпляр MyEmailSender.

Мы сделали еще хуже, теперь PasswordResetHelper зависит от IEmailSender и MyEmailSender, как показано на [рисунке 3-8](#).

**Рисунок 3-8:** Компоненты, которые все равно тесно связаны



Что нам нужно, так это способ получить объекты, которые реализуют данный интерфейс без необходимости создания объекта напрямую. Решение этой проблемы называется *внедрением зависимости* (Dependency injection, DI), известное также как *инверсия управления* (Inversion of Control, IoC).

DI является паттерном, завершающим слабую связь компонентов, которую мы начали с добавления интерфейса IEmailSender в наш простой пример. Вот мы описываем DI, и вы могли бы задаться вопросом, к чему эта суeta, но поймите нас правильно: это важная концепция, которая занимает ключевую позицию в эффективной MVC разработке.

Есть два варианта DI паттерна. Первый состоит в том, что мы удаляем все зависимости от конкретных классов из наших компонентов, в данном случае `PasswordResetHelper`. Это делается путем передачи необходимых интерфейсов в конструктор класса, как показано в [листе 3-4](#).

#### Листинг 3-4: Удаление зависимостей из класса `PasswordResetHelper`

```
public class PasswordResetHelper
{
    private IEmailSender emailSender;

    public PasswordResetHelper(IEmailSender emailSenderParam)
    {
        emailSender = emailSenderParam;
    }

    public void ResetPassword()
    {
        // ...вызов методов интерфейса для конфигурации информации по имейлу...
        emailSender.SendEmail();
    }
}
```

Мы сломали зависимость между `PasswordResetHelper` и `MyEmailSender`: конструктор `PasswordResetHelper` требует объект, который реализует интерфейс `IEmailSender`, но он не знает, или его не волнует, что это за объект и он больше не ответственен за его создание.

Зависимости внедряются в `PasswordResetHelper` во время выполнения, то есть будет создан экземпляр класса, реализующего интерфейс `IEmailSender`, и передан конструктору `PasswordResetHelper` при создании экземпляра. Тут не существует зависимости от времени компиляции между `PasswordResetHelper` и любым классом, который реализует интерфейсы, от которых он зависит.

#### Примечание

*Класс `PasswordResetHelper` требует того, чтобы его зависимости были внедрены с помощью его конструктора – это известно как *constructor injection* (внедрение через конструктор). Мы могли бы также сделать так, чтобы зависимости были внедрены через открытые свойства, что известно как *setter injection* (внедрение через свойства).*

Поскольку с зависимостями работают во время выполнения, мы можем решить, какие реализации интерфейса будут использоваться при запуске приложения: мы можем выбирать между различными компонентами отправки электронной почты или вставлять мок-объекты для тестирования.

## DI пример с MVC

Давайте вернемся к аукционной доменной модели, которую мы создали ранее, и применим к ней DI. Целью является создание класса контроллера, мы назовем его `AdminController`. Он использует репозиторий `MembersRepository`, но без непосредственной связи `AdminController` и `MembersRepository`. Мы начнем с определения интерфейса, который разделит наши два класса – мы назовем его `IMembersRepository` – и изменим класс `MembersRepository` для реализации интерфейса, как показано в [листе 3-5](#).

### Листинг 3-5: Интерфейс IMembersRepository

```
public interface IMembersRepository
{
    void AddMember(Member member);
    Member FetchByLoginName(string loginName);
    void SubmitChanges();
}

public class MembersRepository : IMembersRepository
{
    public void AddMember(Member member)
    {
        /* Реализуй меня */
    }
    public Member FetchByLoginName(string loginName)
    {
        /* Реализуй меня */
    }
    public void SubmitChanges()
    {
        /* Реализуй меня */
    }
}
```

Теперь мы можем написать класс контроллера, который зависит от интерфейса `IMembersRepository`, как показано в [листе 3-6](#).

### Листинг 3-6: Класс AdminController

```
public class AdminController : Controller
{
    IMembersRepository membersRepository;

    public AdminController(IMembersRepository repositoryParam)
    {
        membersRepository = repositoryParam;
    }

    public ActionResult ChangeLoginName(string oldLoginParam, string newLoginParam)
    {
        Member member = membersRepository.FetchByLoginName(oldLoginParam);
        member.LoginName = newLoginParam;
        membersRepository.SubmitChanges();
        // ... теперь будет показано представление
        return View();
    }
}
```

Класс `AdminController` требует реализации интерфейса `IMembersRepository` в качестве параметра конструктора: он будет внедрен во время выполнения, что позволяет `AdminController` работать на экземпляр класса, реализующего интерфейс, не будучи связанным с этой реализацией.

## Использование контейнера внедрения зависимостей

Мы решили вопрос с зависимостью: мы собираемся внедрить наши зависимости в конструкторы наших классов во время выполнения. Но нам все равно нужно решить еще один вопрос – как мы создадим экземпляр конкретной реализации интерфейса без создания зависимостей еще где-нибудь в нашем приложении?

Ответом является контейнер внедрения зависимостей, также известный как IoC контейнер. Это компонент, который действует в качестве посредника между зависимостями, которые требует класс типа `PasswordResetHelper`, и конкретной реализацией этих зависимостей, как `MyEmailSender`.

Мы регистрируем набор интерфейсов или абстрактных типов, которые использует наше приложение, с DI контейнером, и говорим ему, для каких конкретно классов должны быть созданы экземпляры для удовлетворения зависимостей. Таким образом, мы регистрируем интерфейс `IEmailSender` с контейнером и указываем, что должен быть создан экземпляр `MyEmailSender` всякий раз, когда требуется реализация `IEmailSender`. Когда бы нам ни потребовался `IEmailSender`, например, для создания экземпляра `PasswordResetHelper`, мы идем к DI контейнеру, и нам дается реализация класса, который мы зарегистрировали в качестве конкретной реализации по умолчанию этого интерфейса – в данном случае, `MyEmailSender`.

Нам не нужно создавать DI контейнеры самим: есть несколько отличных версий с открытым исходным кодом и свободной лицензией. Один из контейнеров, который нам нравится, называется `Ninject`, и вы можете получить информацию о нем на [www.ninject.org](http://www.ninject.org). Мы познакомим вас с использованием `Ninject` в главе 6.

#### *Совет*

*Microsoft создал свой собственный DI контейнер, который называется Unity. Однако, мы собираемся использовать Ninject, потому что нам он нравится и он демонстрирует способность смешивать и сочетать инструменты при использовании MVC. Если вы хотите получить больше информации о Unity, посетите [unity.codeplex.com](http://unity.codeplex.com).*

---

Роль контейнера DI может показаться простой и тривиальной, но это не тот случай. Хороший DI контейнер, такой как `Ninject`, обладает некоторыми очень умными функциями:

- *Цепочка зависимостей:* Если вы запрашиваете компонент, который имеет свои собственные зависимости (например, параметры конструктора), контейнер также будет удовлетворять эти зависимости. Таким образом, если конструктор для класса `MyEmailSender` требует применения интерфейса `INetworkTransport`, DI контейнер подтвердит реализацию по умолчанию этого интерфейса, передаст его конструктору `MyEmailSender` и вернет результат в виде реализации по умолчанию `IEmailSender`.
- *Управление жизненным циклом объекта:* Если вы запрашиваете компонент более чем один раз, должны ли вы каждый раз получать тот же экземпляр или новый? Хороший DI контейнер позволит вам настроить жизненный цикл компонентов, разрешая выбрать один из предопределенных вариантов, включая *singleton* (тот же экземпляр каждый раз), *transient* (новый экземпляр каждый раз), *instance-per-thread*, *instance-per-HTTP-request*, *instance-from-a-pool* и многие другие.
- *Конфигурация значений параметров конструктора:* Например, если конструктор для нашей реализации интерфейса `INetworkTransport` требует строки `serverName`, вы в состоянии установить значение для нее в конфигурации DI контейнера. Это грубая, но простая система конфигурации, которая удаляет любую необходимость для вашего кода обходить строки подключения, адреса серверов и так далее.

Возможно, вы попытаетесь написать свой собственный DI контейнер. Мы считаем, что это большой экспериментальный проект, если у вас есть время, которое вы можете убить, и вы хотите узнать много нового о C# и .NET отражении (reflection). Если же вы хотите использовать DI

контейнер в MVC приложении для производственных целей, мы рекомендуем вам воспользоваться одним из зарекомендовавших себя DI контейнеров, таким как Ninject.

## Приступим к работе с автоматизированным тестированием

ASP.NET MVC Framework разработан так, чтобы как можно сильнее облегчить создание автоматизированных тестов и использование таких методологий разработки, как TDD, о чём мы поговорим далее в этой главе. ASP.NET MVC обеспечивает идеальную платформу для автоматизированного тестирования, а в Visual Studio есть несколько отличных функций тестирования, которые упрощают создание и выполнение тестов.

В широком смысле, сегодня разработчики веб приложений ставят акцент на двух видах автоматизированного тестирования. Первый вид – это *юнит тестирование* (модульное тестирование), которое является способом определения и проверки поведения отдельных классов (или других небольших кусков кода) в отрыве от остальной части приложения. Второй тип – *интеграционное тестирование*, которое является способом определения и проверки поведения нескольких компонентов, работающих вместе, вплоть до всего веб приложения.

Оба вида тестирования могут быть очень полезны в веб приложениях. Юнит тесты, которые легко создавать и запускать, отлично подходят для работы с алгоритмами, бизнес логикой и другой бэкэнд инфраструктурой.

Значение интеграционного тестирования заключается в том, что при помощи него можно смоделировать, как пользователь будет взаимодействовать с пользовательским интерфейсом, и оно может охватывать весь стек технологий, которые использует приложение, в том числе веб сервера и базы данных. Интеграционное тестирование, как правило, лучше всего проявляется на выявлении новых ошибок, возникших в старых функциях; это известно как *регрессионное тестирование*.

### Юнит тестирование

В мир .NET вы создаете отдельный тестовый проект в решении Visual Studio для хранения *тестовых фикстур*. Этот проект будет создан при первом добавлении модульного теста, или он может быть создан автоматически при использовании шаблона проекта MVC. *Тестовая фикстура* – это C# класс, который определяет набор методов – один метод для каждого вида поведения, что вы хотите проверить. Тестовый проект может содержать несколько классов тестовых фикстур.

#### Примечание

*Мы покажем вам, как создать тестовый проект и заполнить его юнит тестами в главе 6. Цель этой главы заключается в простом введении в понятие модульного тестирования и в том, чтобы дать вам представление, как выглядит тестовая фикстура и как она используется.*

---

В [листеинге 3-7](#) содержится пример тестовой фикстуры, которая проверяет поведение метода AdminController.ChangeLoginName, что мы определили в [листеинге 3-6](#).

### Листинг 3-7: Пример тестовой фикстуры

```
[TestClass]
public class AdminControllerTest
{
    [TestMethod]
    public void CanChangeLoginName()
    {
        // Arrange (устанавливается сценарий)
        Member bob = new Member() { LoginName = "Bob" };
        FakeMembersRepository repositoryParam = new FakeMembersRepository();
        repositoryParam.Members.Add(bob);
        AdminController target = new AdminController(repositoryParam);
        string oldLoginParam = bob.LoginName;
        string newLoginParam = "Anastasia";
        // Act (проводится операция)
        target.ChangeLoginName(oldLoginParam, newLoginParam);
        // Assert (проверяется результат)
        Assert.AreEqual(newLoginParam, bob.LoginName);
        Assert.IsTrue(repositoryParam.DidSubmitChanges);
    }
    private class FakeMembersRepository : IMembersRepository
    {
        public List<Member> Members = new List<Member>();
        public bool DidSubmitChanges = false;
        public void AddMember(Member member)
        {
            throw new NotImplementedException();
        }
        public Member FetchByLoginName(string loginName)
        {
            return Members.First(m => m.LoginName == loginName);
        }
        public void SubmitChanges()
        {
            DidSubmitChanges = true;
        }
    }
}
```

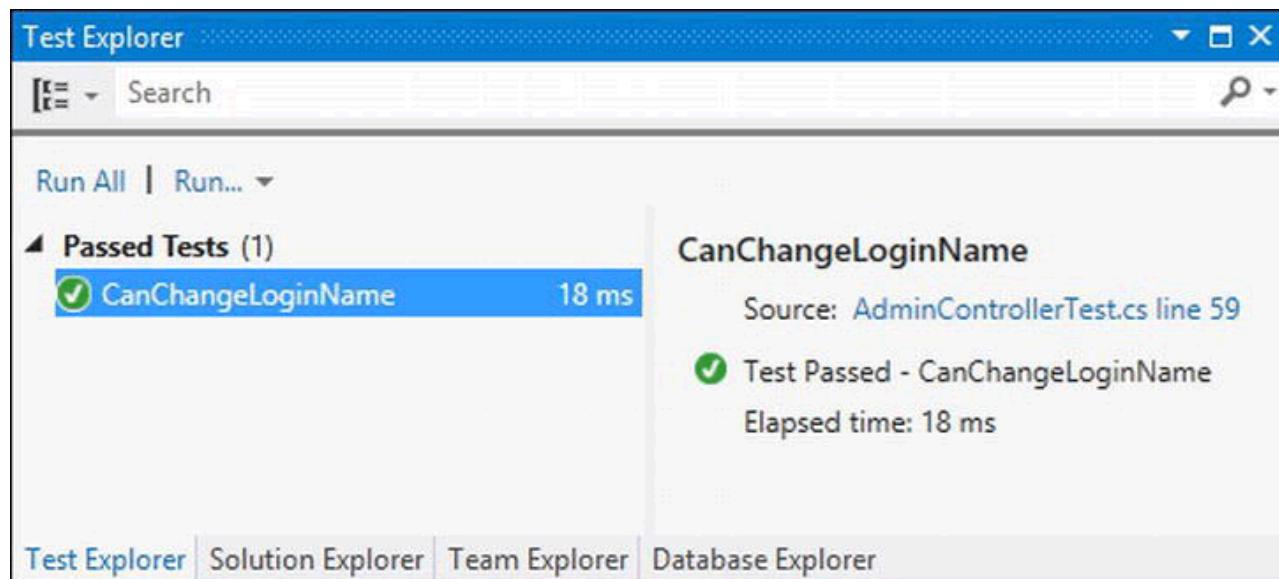
Тестовая фикстура – это метод `CanChangeLoginName`. Обратите внимание, что в методе присутствует атрибут `TestMethod`, и что в классе, к которому он принадлежит, `AdminControllerTest`, есть атрибут `TestClass`: это способ, как Visual Studio находит тестовую фикстуру.

Метод `CanChangeLoginName` следует паттерну, известному как *arrange/act/assert* (A/A/A). *Arrange* относится к созданию условий для теста, *act* относится к выполнению теста, а *assert* относится к проверке того, что получен требуемый результат. Будучи последовательными в создании структуры ваших методов юнит тестов, старайтесь делать их легкими для прочтения: вы по достоинству это оцените, когда ваш проект будет содержать сотни юнит тестов.

Тестовая фикстура в [листе 3-7](#) использует специальную тестовую симулирующую реализацию интерфейса `IMembersRepository` для имитации определенных условий, в данном случае, когда в репозитории есть один член, `Bob`. Создание имитационного репозитория и `Member` сделано в разделе *arrange* теста.

Далее, вызывается testируемый метод `AdminController.ChangeLoginName`. Это часть *act* теста. И наконец, мы проверяем результаты с помощью пары вызовов `Assert` (это *assert* часть теста). Мы запускаем тест с помощью Visual Studio меню `Test` и получаем визуальное представление о том, как выполняются тесты, что показано на [рисунке 3-9](#).

**Рисунок 3-9:** Визуальное представление о прогрессе юнит теста



Если тестовая фикстура выполняется без необработанных исключений, и все выражения `Assert` срабатывают без проблем, в окне `Test Results` показана галочка на зеленом фоне – если же что-то пошло не так, то вы увидите красный фон и информацию о том, что не верно.

#### Примечание

*Вы можете видеть, как использование DI помогло нам с модульным тестированием. Мы смогли создать симулирующую реализацию репозитория и внедрить ее в контроллер, чтобы создать специфичный сценарий. Мы большие поклонники DI, и это одна из причин.*

Может показаться, что мы приложили массу усилий, чтобы проверить простой метод, но вам не потребуется гораздо большего количества кода, чтобы проверить нечто намного более сложное. Если вы посчитаете, что лучше пропускать небольшие тесты, подобные этому, поразмыслите над тем, что такие тестовые фикстуры помогают обнаруживать ошибки, которые иногда могут быть скрыты в более сложных тестах.

По мере прочтения книги вы увидите примеры более сложных и четких тестов. Одно из улучшений, которое мы можем сделать, заключается в том, что мы можем устраниć тестовые симулирующие классы, как `FakeMembersRepository`, с помощью mock-объектов. Мы покажем вам, как это сделать, в главе 6.

#### Использование TDD и процесс Red-Green-Refactor

В TDD можно использовать юнит тесты, чтобы создавать код. Это может показаться вам странной концепцией, если вы привыкли к тестированию после окончания кодирования, но в таком подходе есть смысл. Ключевым понятием здесь является рабочий процесс разработки, который называются red-green-refactor (красный-зеленый-рефакторинг). Вот как это работает:

- Определяем, что нам нужно добавить новую функцию или метод в приложение.
- Пишем тест, который будет проверять поведение новой функции, когда она написана.
- Запускаем тест и получаем галочку на красном фоне.
- Пишем код, который реализует новую функцию.
- Снова запускаем тест и корректируем код, пока не появится зеленый фон.

- Если требуется, оптимизируем код (проводим рефакторинг), например, реорганизация выражений, переименование переменных и так далее.
- Запускаем тест, чтобы подтвердить, что изменения не повлияли на поведение дополнений.

Этот рабочий процесс повторяется для каждой функции, которую вы добавляете. Давайте рассмотрим пример, чтобы вы могли понять, как это работает. Давайте представим, что нам нужно следующее поведение: возможность добавлять ставку для предмета, но только если эта ставка выше, чем все предыдущие ставки для этого предмета. Во-первых, мы добавим метод-заглушку для класса `Item`, как показано в [листинге 3-8](#).

### Листинг 3-8: Добавление метода-заглушки для класса `Item`

```
using System;
using System.Collections.Generic;
namespace TheMVCPattern.Models
{
    public class Item
    {
        public int ItemID { get; private set; } // Уникальный ключ
        public string Title { get; set; }
        public string Description { get; set; }
        public DateTime AuctionEndDate { get; set; }
        public IList<Bid> Bids { get; private set; }
        public void AddBid(Member memberParam, decimal amountParam)
        {
            throw new NotImplementedException();
        }
    }
}
```

Очевидно, что метод `AddBid`, выделенный жирным шрифтом, не отображает требуемое поведение, но пусть это вас не останавливает – ключом к TDD является проверка на правильное поведения до реализации этой функции. Мы собираемся протестировать три различных аспекта поведения, которое мы хотим реализовать:

- Когда нет ставок, может быть добавлено любое значение.
- Когда ставки есть, может быть добавлено более высокое значение.
- Когда ставки есть, более низкое значение не может быть добавлено.

Что бы сделать это, мы создадим три тестовых метода, как показано в [листинге 3-9](#).

### Листинг 3-9: Три тестовые фикстуры

```
[TestMethod()]
public void CanAddBid()
{
    // Arrange - устанавливается сценарий
    Item target = new Item();
    Member memberParam = new Member();
    Decimal amountParam = 150M;
    // Act - выполняется тест
    target.AddBid(memberParam, amountParam);
    // Assert - проверяется поведение
    Assert.AreEqual(1, target.Bids.Count());
    Assert.AreEqual(amountParam, target.Bids[0].BidAmount);
}

[TestMethod()]
[ExpectedException(typeof(InvalidOperationException))]
public void CannotAddLowerBid()
{
    // Arrange
```

```

Item target = new Item();
Member memberParam = new Member();
Decimal amountParam = 150M;
// Act
target.AddBid(memberParam, amountParam);
target.AddBid(memberParam, amountParam - 10);
}

[TestMethod()]
public void CanAddHigherBid() {
    // Arrange
    Item target = new Item();
    Member firstMember = new Member();
    Member secondMember = new Member();
    Decimal amountParam = 150M;
    // Act
    target.AddBid(firstMember, amountParam);
    target.AddBid(secondMember, amountParam + 10);
    // Assert
    Assert.AreEqual(2, target.Bids.Count());
    Assert.AreEqual(amountParam + 10, target.Bids[1].BidAmount);
}

```

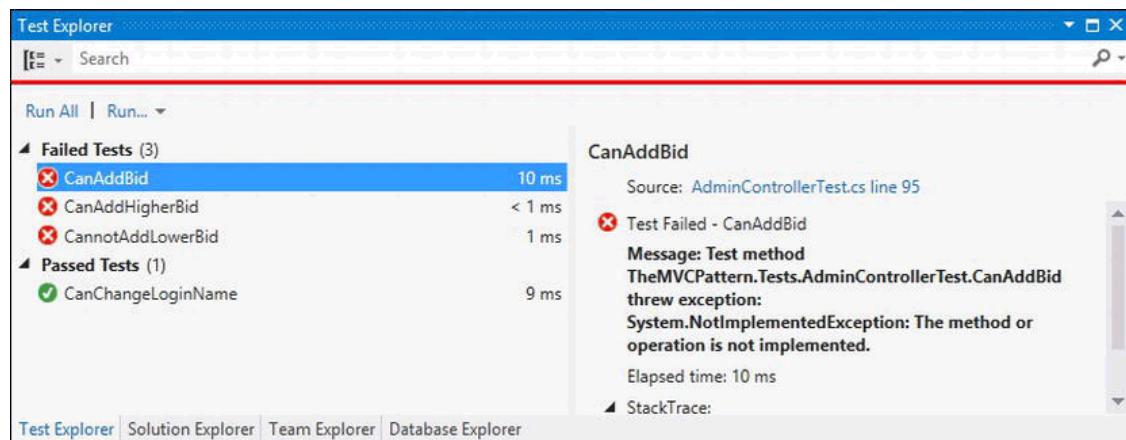
Вы видите, что мы создали модульный тест для каждого из типов поведения, которые мы хотим видеть. Тестовые методы следуют паттерну `arrange/act/assert` для создания, тестирования и проверки одного из аспектов общего поведения. Метод `CannotAddLowerBid` не имеет утвержденной части в теле метода, поскольку успешный тест – это исключение, которые мы утверждаем, применяя атрибут `ExpectedException` тестового метода.

#### Примечание

*Обратите внимание, как тест, который мы выполняем в методе юнит теста `CannotAddLowerBid` будут формировать реализацию метода `AddBid`. Мы валидируем результат теста, гарантируя, что есть исключение и что оно является экземпляром `System.InvalidOperationException`. Написание юнит тестов перед написанием кода может помочь вам подумать о том, какие результаты должны быть выражены, прежде чем увязнуть в реализации.*

Как и следовало ожидать, все эти тесты не сработают, когда мы их запустим, как показано на [рисунке 3-10](#).

**Рисунок 3-10:** Первый запуск юнит тестов



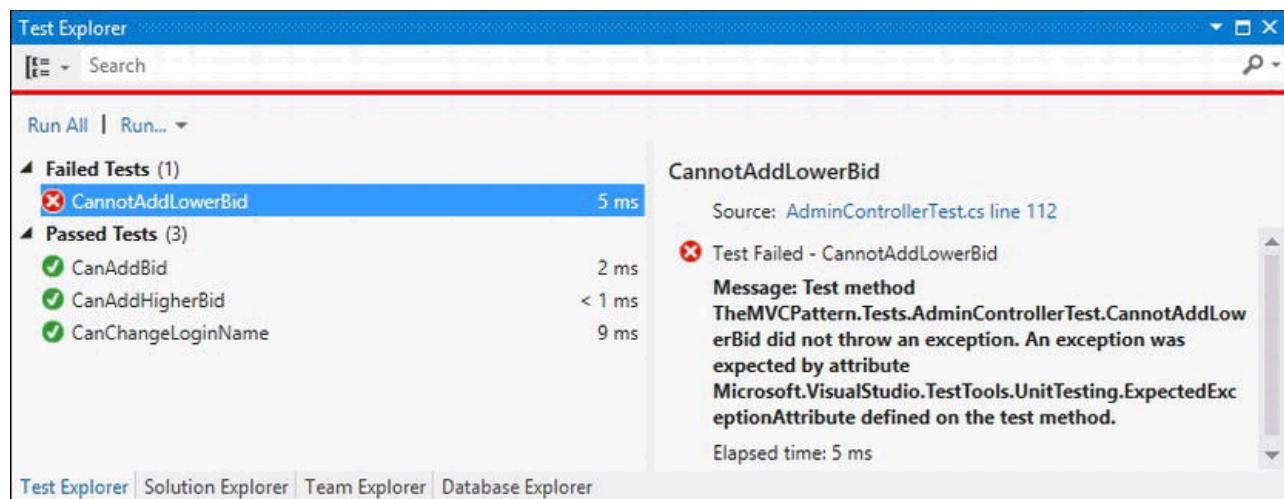
Теперь может быть реализован наш первый переход к методу AddBid, как показано в [листинге 3-10](#).

### Листинг 3-10: Реализация метода AddBid

```
using System;
using System.Collections.Generic;
namespace TheMVCPattern.Models
{
    public class Item
    {
        public int ItemID { get; private set; } // Уникальный ключ
        public string Title { get; set; }
        public string Description { get; set; }
        public DateTime AuctionEndDate { get; set; }
        public IList<Bid> Bids { get; set; }
        public Item()
        {
            Bids = new List<Bid>();
        }
        public void AddBid(Member memberParam, decimal amountParam)
        {
            Bids.Add(new Bid()
            {
                BidAmount = amountParam,
                DatePlaced = DateTime.Now,
                Member = memberParam
            });
        }
    }
}
```

Мы добавили первоначальную реализацию метода `AddBid` классу `Item`. Мы также добавили простой конструктор, поэтому мы можем создавать экземпляры `Item` и знать, что коллекция объектов `Bid` правильно инициализирована. Новый запуск юнит тестов покажет более хорошие результаты, как представлено на [рисунке 3-11](#).

### Рисунок 3-11: Новый запуск юнит тестов



Два из трех модульных тестов были успешными. Для того, который не сработал – `CannotAddLowerBid` – мы не добавили никаких проверок, чтобы убедиться, что ставка выше, чем предыдущие ставки по данному лоту. Мы должны изменить нашу реализацию, чтобы вставить эту логику, как показано в [листинге 3-11](#).

**Листинг 3-11:** Улучшение реализации метода `AddBid`

```
using System;
using System.Collections.Generic;
using System.Linq;
namespace TheMVCPattern.Models
{
    public class Item
    {
        public int ItemID { get; private set; } // Уникальный ключ
        public string Title { get; set; }
        public string Description { get; set; }
        public DateTime AuctionEndDate { get; set; }
        public IList<Bid> Bids { get; set; }
        public Item()
        {
            Bids = new List<Bid>();
        }
        public void AddBid(Member memberParam, decimal amountParam)
        {
            if (Bids.Count() == 0 || amountParam > Bids.Max(e => e.BidAmount))
            {
                Bids.Add(new Bid()
                {
                    BidAmount = amountParam,
                    DatePlaced = DateTime.Now,
                    Member = memberParam
                });
            }
            else
            {
                throw new InvalidOperationException("Bid amount too low");
            }
        }
    }
}
```

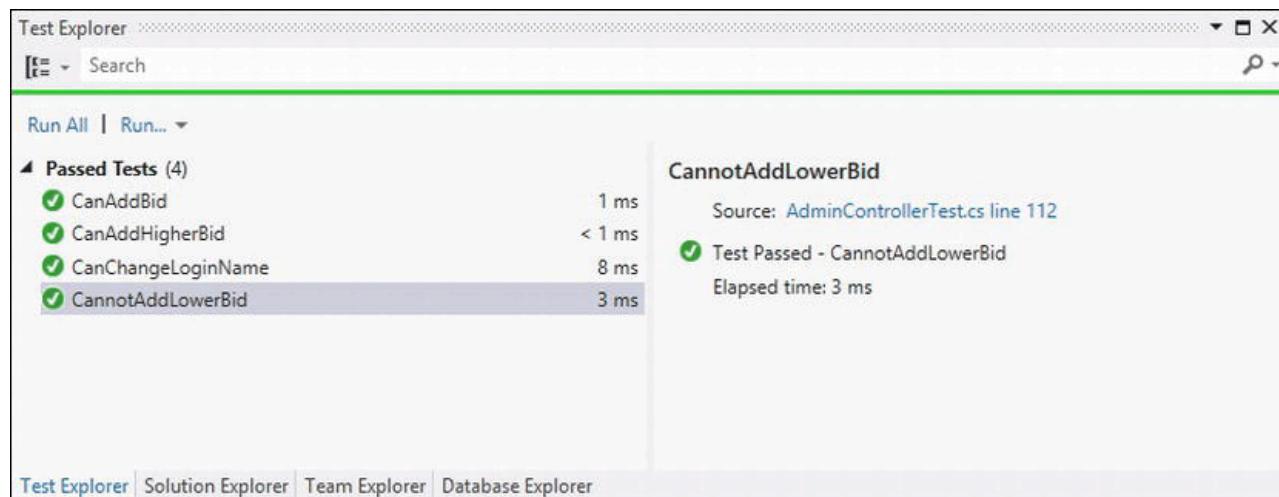
Вы видите, что мы выразили условие ошибки таким образом, чтобы удовлетворить модульный тест, который мы написали прежде, чем начали кодирование, то есть мы бросаем `InvalidOperationException`, если ставка слишком низкая.

**Примечание**

*Мы использовали **Language Integrated Query (LINQ)**, чтобы убедиться, что ставка действительна. Не волнуйтесь, если вы не знакомы с LINQ или лямбда-выражениями, которые мы использовали (обозначение =>): мы познакомим вас с функциями C#, которые имеют важное значение для разработки с MVC, в главе 6.*

Каждый раз, когда мы меняем реализацию метода `AddBid`, мы заново запускаем наши юнит тесты. Результаты показаны на [рисунке 3-12](#).

**Рисунок 3-12:** Успешные результаты юнит тестов



Удача! Мы реализовали новую функцию так, что она проходит все юнит тесты. Но нам еще нужно убедиться, что наши тесты действительно проверяют все аспекты поведения или функций, которые мы реализуем. Если это так, то мы все сделали. Если же нет, то мы добавим больше тестов и повторим цикл, и будем продолжать, пока не станем уверены, что у нас есть полный набор тестов и та реализация, которая их все успешно проходит.

Этот цикл является сущностью TDD. Существует множество причин, чтобы рекомендовать его как стиль программирования: не в последнюю очередь потому, что он заставляет программиста подумать о том, как изменение или расширение должно вести себя, *перед* началом кодирования. У вас всегда есть четкое представление о том, что происходит. И если у вас есть юнит тесты, которые покрывают остальную часть вашего приложения, вы будете уверены, что ваши дополнения не изменили поведение других местах.

### Религия модульного тестирования

Если вы в настоящее время не проводите модульного тестирования кода, вы можете посчитать этот процесс ненужным и разрушительным: больше писанины, больше тестирования, большее количество итераций. Если вы *используете* юнит тесты, вы уже понимаете разницу: меньше ошибок, лучше разработанное программное обеспечение и меньше сюрпризов, когда вы вносите изменения.

Переход от «нетестировщика» к тестировщику может быть жестким: это обозначает, что принятие новой привычки и привыкание к ней может быть довольно долгим процессом, прежде чем вы получите выгоду. Наши первые попытки заняться тестированием провалились из-за неожиданного изменения в сроках, ведь сложно себя убедить делать что-то дополнительно, когда времени мало.

Мы оба стали приверженцами модульного тестирования и убеждены, что это отличный стиль разработки. ASP.NET MVC является идеальным кандидатом для принятия модульного тестирования, если вы никогда не пробовали его раньше или если вы попробовали и отказались. Команда Microsoft сделала модульное тестирование невероятно легким, разделяя основные классы от базовой технологии. Это означает, мы можем создавать *mock-объекты* для ключевых функций и тестировать ситуации, которые было бы невероятно трудно воспроизвести в ином случае. Мы покажем вам в этой книге примеры модульного тестирования MVC приложений. Мы рекомендуем вам следовать с нами и попробовать модульное тестирование для своих приложений.

## Интеграционное тестирование

Для веб приложений наиболее распространенным подходом интеграционного тестирования является *автоматизация UI*, что обозначает моделирование или автоматизацию веб браузера для осуществления всего стека технологии приложения путем воспроизведения действий, которые может выполнить пользователь, таких как нажатие кнопок, переходы по ссылкам и отправка формы.

Вот два самых известных инструмента с открытым исходным кодом по автоматизации браузера для .NET разработчиков:

- *Selenium RC* (<http://seleniumhq.org/>), состоящий из "серверного" Java приложения. Он может отправлять команды автоматизации для Internet Explorer, Firefox, Safari или Opera. Также он поддерживает .NET, Python, Ruby и так далее, так что вы можете писать тестовые скрипты на языке по вашему выбору. Selenium является мощным и зрелым инструментом, его единственный недостаток состоит в том, что вы должны запускать его Java сервер.
- *WatiN* (<http://watin.org>), .NET библиотека, которая может посыпать команды для автоматизации Internet Explorer или Firefox. Его API не является столь мощным, как Selenium, но он удобно обрабатывает наиболее распространенные сценарии и прост в настройке: вам нужно ссылаться только на одну DLL.

Интеграционное тестирование является идеальным дополнением к модульному тестированию. Хотя модульное тестирование хорошо подходит для проверки поведения отдельных компонентов на сервере, интеграционное тестирование позволяет создавать тесты, которые ориентированы на клиента, воссоздавая действия пользователя. В результате это может выявить проблемы, которые происходят от взаимодействия между компонентами – отсюда и термин *интеграционное тестирование*. И поскольку интеграционное тестирование веб приложения осуществляется через браузер, вы можете проверить, что JavaScript работает так, как вам надо, что очень трудно сделать, используя модульное тестирование.

Конечно же, есть и некоторые недостатки – интеграционное тестирование занимает больше времени. Больше времени требуется для создания тестов и больше времени на их выполнение. Интеграционные тесты могут быть хрупкими: например, если вы измените атрибут `id` элемента, который проверяется в teste, тест может не сработать (и обычно не срабатывает).

В результате того, что требуется дополнительное время и усилия, интеграционное тестирование часто проводится на ключевых этапах проекта, возможно, после еженедельной проверки кода или когда завершены основные функциональные блоки. Интеграционное тестирование ничуть не хуже модульного тестирования, и оно может выявить проблемы, которые модульное тестирование выявить не может. Время, необходимое для установки и запуска интеграционного тестирования стоит того, и мы советуем вам добавить его в процесс разработки.

Мы не собираемся рассматривать интеграционное тестирование в этой книге. Это не потому, что мы не думаем, что оно не полезно – наоборот, именно поэтому мы настоятельно рекомендуем вам добавить его в свой процесс разработки – а поскольку оно выходит за рамки интереса данной книги. ASP.NET MVC фреймворк был разработан специально для того, чтобы сделать модульное тестирование легким и простым, и мы должны включить модульное тестирование в эту книгу, чтобы дать вам полные знания о том, как построить хорошее MVC приложение. Интеграционное тестирование – это отдельное искусство и то, что верно при выполнении интеграционного тестирования для любого веб приложения верно и для MVC.

# Резюме

В этой главе мы показали вам архитектурный паттерн MVC и сравнили его с некоторыми другими паттернами, которые вы, возможно, видели или слышали. Мы обсудили значимость доменной модели и создали простой ее пример. Мы также показали внедрение зависимости, что позволяет отделять компоненты друг от друга для обеспечения строгого разделения различных частей нашего приложения. Мы продемонстрировали некоторые простые модульные тесты, и вы увидели, как разделенные компоненты и внедрение зависимостей помогают сделать модульное тестирование простым и легким. Попутно мы продемонстрировали наше увлечение TDD и показали, как мы пишем юнит тесты, прежде чем писать код приложения. Наконец, мы затронули интеграционное тестирование и сравнили его с модульным тестированием.

# Основные особенности языка

C# является многофункциональным языком, но не все программисты знакомы со всеми функциональными возможностями, на которые мы будем опираться в этой книге. В данной главе мы рассмотрим функции языка C#, которые хорошо должен знать MVC программист.

Мы предоставим только краткий обзор каждой функции. Если вы хотите более глубокого охвата C# или LINQ, для вас могут представлять интерес три книги Адама. Для полного руководства по C# попробуйте *Introducing Visual C#*; для углубленного изучения LINQ посмотрите *Pro LINQ in C#*; а для детального изучения .NET поддержки асинхронного программирования обратите внимание на *Pro .Net Parallel Programming in C#*. Все эти книги изданы Apress.

## Создание проекта-примера

Для демонстрации возможностей языка в этой части книги, мы создали новое Visual Studio проект ASP.NET MVC 4 Web Application под названием `LanguageFeatures` и выбрали опцию шаблона `Empty`. Функции не являются специфическими для MVC, но Visual Studio Express 2012 для веб не поддерживает создание проектов, которые могут выводить на консоль, так что вам придется создать MVC приложение, если вы хотите следовать за примерами.

Нам понадобится простой контроллер для демонстрации этих особенностей языка, поэтому мы создали файл `HomeController.cs` в папке `Controllers`, используя технические приемы, которые мы показали вам в [главе 2](#). Вы можете увидеть начальное содержимое контроллера `Home` в [листинге 4-1](#).

**Листинг 4-1:** Начальное содержимое контроллера `Home`

```
using LanguageFeatures.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
namespace LanguageFeatures.Controllers
{
    public class HomeController : Controller
    {
        public string Index()
        {
            return "Navigate to a URL to show an example";
        }
    }
}
```

Для каждого примера мы будем создавать методы действия, поэтому результатом метода действия `Index` является простое сообщение, чтобы не усложнять проект. Для отображения результатов наших методов действия мы добавили представление с именем `Result.cshtml` в папку `Views/Home`. Вы можете посмотреть содержимое файла представления в [листинге 4-2](#).

**Листинг 4-2:** Содержимое файла представления `Result`

```
@model String
@{
Layout = null;
}
<!DOCTYPE html>
<html>
<head>
```

```

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

```

Вы видите, что это строго типизированное представление, где типом модели является `String`: это не сложные примеры, и мы можем легко представить результаты в виде простых строк.

## Использование автоматически реализуемых свойств

Возможности свойств C# позволяют представить часть данных из класса таким образом, что мы получаем чистые данные, независимо от того, как они были установлены и получены. В [листе 4-3](#) содержится простой пример класса `Product`, который мы добавили в папку `Models` проекта `LanguageFeatures`.

### Листинг 4-3: Определение свойства

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
namespace LanguageFeatures.Models
{
    public class Product
    {
        private string name;
        public string Name
        {
            get { return name; }
            set { name = value; }
        }
    }
}

```

Свойство `Name` выделено жирным шрифтом. Выражения в блоке кода `get` выполняются при чтении значения свойства, а выражения в блоке кода `set` выполняются, когда значение присваивается свойству (специальная переменная `value` представляет присвоенное значение). Свойство используется другими классами, как показано в [листе 4-4](#), который представляет метод действия `AutoProperty`, что мы добавили к контроллеру `Home`.

### Листинг 4-4: Использование свойства

```

using LanguageFeatures.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
namespace LanguageFeatures.Controllers
{
    public class HomeController : Controller
    {
        public string Index()
        {

```

```

        return "Navigate to a URL to show an example";
    }
    public ViewResult AutoProperty()
    {
        // создается новый объект Product
        Product myProduct = new Product();
        // устанавливается значение свойства
        myProduct.Name = "Kayak";
        // читается свойство
        string productName = myProduct.Name;
        // генерируется представление
        return View("Result",
            (object)String.Format("Product name: {0}", productName));
    }
}
}
}

```

Вы видите, что значение свойства читается и устанавливается, как обычное поле. Использование свойств предпочтительнее использования полей, потому что вы можете менять выражения в блоках `get` и `set` без необходимости изменения всех классов, которые зависят от свойства.

#### *Совет*

*Вы заметили, что в [листинге 4-4](#) мы привели второй аргумент для метода `View` к типу `object`. Это потому что метод `View` перегружен, то есть принимает два аргумента `String`, и это имеет другое значение, чем принимать `String` и `object`. Чтобы избежать не того вызова, мы явно привели второй аргумент. Мы вернемся к методу `View` и всем его перегрузкам в главе 18.*

---

Вы можете увидеть результат выполнения этого примера, запустив проект и перейдя к `/Home/AutoProperty` (это направлено на вызов метода действия `AutoProperty`, и мы получаем шаблон для тестирования каждого примера в этой главе). Поскольку мы передали из метода действия в представление только строку, мы покажем вам результат в виде текста, а не в виде скриншота. Результат [листинга 4-4](#):

**Product name: Kayak**

Все вроде бы хорошо, но это станет утомительным, когда у нас появится класс, имеющий много свойств, которые служат связующим звеном доступа к полю. И в итоге мы получим нечто излишне громоздкое, как показано в [листинге 4-5](#). Тут представлено, как эти свойства выглядят в файле `Product.cs`.

#### **Листинг 4-5:** Много определений свойств

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
namespace LanguageFeatures.Models
{
    public class Product
    {
        private int productID;
        private string name;
        private string description;
        private decimal price;
        private string category;
        public int ProductID
        {
            get { return productID; }

```

```

        set { productID = value; }
    }
    public string Name
    {
        get { return name; }
        set { name = value; }
    }
    public string Description
    {
        get { return description; }
        set { description = value; }
    }
    //...и так далее...
}
}

```

Мы хотим гибкости свойств, но на данный момент нам не нужны пользовательские выражения получения и установки. Решением является *автоматически реализуемое свойство*, также известное как *автоматическое свойство*. При помощи автоматического свойства вы можете создать шаблон свойства, поддерживающего поле, без определения поля или указания кода для выражений получения и установки, как показано в [листеинге 4-6](#).

#### **Листинг 4-6:** Использование автоматически реализуемых свойств

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
namespace LanguageFeatures.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; }
    }
}

```

Есть несколько пунктов, на которые стоит обратить внимание при использовании автоматических свойств. Первое, мы не определяем тело для выражений получения и установки. Второе, мы не определяем поле, поддерживающее свойство. Все это сделает для нас C# компилятор, когда мы построим наш класс. Использование автоматических свойств ничем не отличается от использования обычных свойств; код в методе действия из [листеинга 4-4](#) будет работать без каких-либо изменений.

С помощью автоматических свойств мы экономим время на наборе текста, создаем код, который легче читать, и при этом сохраняем ту гибкость, которую предоставляют свойства. Если наступит момент, когда нам нужно будет изменить способ реализации свойства, мы сможем вернуться к обычному формату свойства. Давайте представим, что мы должны изменить способ реализации свойства `Name`, как показано в [листеинге 4-7](#).

#### **Листинг 4-7:** Возвращение от автоматического к обычному свойству

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
namespace LanguageFeatures.Models
{
    public class Product
    {

```

```

private string name;
public int ProductID { get; set; }
public string Name
{
    get
    {
        return ProductID + name;
    }
    set
    {
        name = value;
    }
}
public string Description { get; set; }
public decimal Price { get; set; }
public string Category { set; get; }
}
}

```

**Примечание**

*Обратите внимание, что мы должны реализовать выражения получения и установки, чтобы вернуться к обычному свойству. C# не поддерживает смешивание автоматического и обычного стиля выражений получения и установки в одном свойстве.*

---

## Использование инициализаторов объектов и коллекций

Другой утомительной задачей программирования является создание нового объекта, а затем присвоение значений свойствам, как показано в [листинге 4-8](#), который демонстрирует добавление метода действия CreateProduct в контроллер Home.

**Листинг 4-8:** Создание и инициализация объекта со свойствами

```

using LanguageFeatures.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
namespace LanguageFeatures.Controllers
{
    public class HomeController : Controller
    {
        public string Index()
        {
            return "Navigate to a URL to show an example";
        }
        public ViewResult AutoProperty()
        {
            // ...выражения опущены для краткости...
        }
        public ViewResult CreateProduct()
        {
            // создание нового объекта Product
            Product myProduct = new Product();
            // установка значений свойств
            myProduct.ProductID = 100;
            myProduct.Name = "Kayak";
        }
    }
}

```

```
    myProduct.Description = "A boat for one person";
    myProduct.Price = 275M;
    myProduct.Category = "Watersports";
    return View("Result",
        (object)String.Format("Category: {0}", myProduct.Category));
}
}
```

Мы должны пройти три этапа, чтобы создать объект `Product` и получить результат: создать объект, установить значения параметров, а затем вызвать метод `View`, чтобы мы могли увидеть результат через представление. К счастью, мы можем использовать функцию *инициализации объекта*, которая позволяет создавать и заполнять экземпляр объекта `Product` за один шаг, как показано в [листинге 4-9](#).

**Листинг 4-9:** Использование функции инициализации объекта

```
public ViewResult CreateProduct() {
    // создание и заполнение нового объекта Product
    Product myProduct = new Product {
        ProductID = 100, Name = "Kayak",
        Description = "A boat for one person",
        Price = 275M, Category = "Watersports"
    };
    return View("Result", (object)String.Format("Category: {0}", myProduct.Category));
}
```

Фигурные скобки (`{}`) после вызова названия `Product` образуют *инициализатор*, который мы используем для передачи значений параметров как часть процесса создания. Эта же функция позволяет нам инициализировать содержимое коллекций и массивов как часть процесса создания, что показано в [листинге 4-10](#).

#### Листинг 4-10: Инициализация коллекций и массивов

```
using LanguageFeatures.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
namespace LanguageFeatures.Controllers
{
    public class HomeController : Controller
    {
        public string Index()
        {
            return "Navigate to a URL to show an example";
        }
        // ...другие методы действия опущены для краткости...
        public ViewResult CreateCollection()
        {
            string[] stringArray = { "apple", "orange", "plum" };
            List<int> intList = new List<int> { 10, 20, 30, 40 };
            Dictionary<string, int> myDict = new Dictionary<string, int> {
                { "apple", 10 }, { "orange", 20 }, { "plum", 30 }
            };
            return View("Result", (object)stringArray[1]);
        }
    }
}
```

В листинге показано, как создать и инициализировать массив и два класса из общей библиотеки коллекции. Эта функция располагает удобным синтаксисом – она просто делает C# более приятным в использовании, но не имеет никакого другого воздействия или выгоды.

## Использование методов расширения

Методы расширения представляют собой удобный способ добавления методов в классы, которые вам не принадлежат и поэтому не могут быть изменены напрямую. В [листе 4-11](#) показан класс ShoppingCart, который мы добавили в папку Models и который представляет собой коллекцию объектов Product.

### Листинг 4-11: Класс ShoppingCart

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
namespace LanguageFeatures.Models
{
    public class ShoppingCart
    {
        public List<Product> Products { get; set; }
    }
}
```

Это очень простой класс, который действует как оболочка вокруг List объектов Product ( нам нужен простой класс для данного примера). Предположим, что нам нужна возможность определять общую стоимость объектов Product в классе ShoppingCart, но мы не можем изменить сам класс, например, потому что он получен от третьей стороны и у нас нет исходного кода. К счастью, мы можем использовать метод расширения для получения необходимого функционала. В [листе 4-12](#) показан класс MyExtensionMethods, который мы также добавили в папку Models.

### Листинг 4-12: Определение метода расширения

```
namespace LanguageFeatures.Models
{
    public static class MyExtensionMethods
    {
        public static decimal TotalPrices(this ShoppingCart cartParam)
        {
            decimal total = 0;
            foreach (Product prod in cartParam.Products)
            {
                total += prod.Price;
            }
            return total;
        }
    }
}
```

Ключевое слово this перед первым параметром отмечает TotalPrices как расширенный метод. Первый параметр говорит .NET, к какому классу можно применять метод расширения, в нашем случае к ShoppingCart. Мы можем обратиться к экземпляру ShoppingCart, к которому был применен расширенный метод, с помощью параметра cartParam. Наш метод перечисляет объекты Product в ShoppingCart и возвращает сумму свойства Product.Price. В [листе 4-13](#) показано, как мы применяем метод расширения в новом методе действия UseExtension, который мы добавили контроллеру Home.

## Примечание

*Методы расширения не позволяют обойти правила доступа, которые определяют классы для своих методов, полей и свойств. Вы можете расширить функционал класса с помощью расширенного метода, но только используя членов класса, к которым у вас так или иначе есть доступ.*

### Листинг 4-13: Применение метода расширения

```
using System.Linq;
using System.Web;
using System.Web.Mvc;
namespace LanguageFeatures.Controllers
{
    public class HomeController : Controller
    {
        public string Index()
        {
            return "Navigate to a URL to show an example";
        }
        // ...другие методы действия опущены для краткости...
        public ViewResult UseExtension()
        {
            // создание и заполнение ShoppingCart
            ShoppingCart cart = new ShoppingCart
            {
                Products = new List<Product> {
                    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}
                }
            };
            // получение общей стоимости продуктов в корзине
            decimal cartTotal = cart.TotalPrices();
            return View("Result",
                (object)String.Format("Total: {0:c}", cartTotal));
        }
    }
}
```

Ключевым выражением является вот это:

```
...
decimal cartTotal = cart.TotalPrices();
...
```

Как вы видите, мы называем метод `TotalPrices` для объекта `ShoppingCart`, как будто это часть класса `ShoppingCart`, даже если это расширенный метод, определенный другим классом. .NET найдет расширенные методы, если они находятся в рамках текущего класса, что обозначает, что они являются частью одного и того же пространства имен или находятся в пространстве имен, которое является предметом выражения `using`. Вот результат использования метода действия `UseExtension`:

```
Total: $378.40
```

## Применение методов расширения к интерфейсу

Мы также можем создать расширенные методы, применяемые к интерфейсу, который позволяет вызвать метод расширения для всех классов, реализующих этот интерфейс. В [листе 4-14](#) показан класс `ShoppingCart`, обновленный для реализации интерфейса `IEnumerable<Product>`.

#### Листинг 4-14: Реализация интерфейса в классе ShoppingCart

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Web;
namespace LanguageFeatures.Models
{
    public class ShoppingCart : IEnumerable<Product>
    {
        public List<Product> Products { get; set; }
        public IEnumerator<Product> GetEnumerator()
        {
            return Products.GetEnumerator();
        }
        IEnumerable IEnumerable.GetEnumerator()
        {
            return GetEnumerator();
        }
    }
}
```

Теперь мы можем обновить наш метод расширения, чтобы он работал с **IEnumerable<Product>**, как показано в [листе 4-15](#).

#### Листинг 4-15: Метод расширения, который работает с интерфейсом

```
using System.Collections.Generic;
namespace LanguageFeatures.Models
{
    public static class MyExtensionMethods
    {
        public static decimal TotalPrices(this IEnumerable<Product> productEnum)
        {
            decimal total = 0;
            foreach (Product prod in productEnum)
            {
                total += prod.Price;
            }
            return total;
        }
    }
}
```

Первый тип параметра изменился на **IEnumerable<Product>**, это обозначает, что цикл `foreach` в теле метода работает непосредственно с объектами `Product`. В противном случае метод расширения остается неизменным. Переход на интерфейс означает, что мы можем рассчитать общую стоимость объектов `Product`, перечисленных любым **IEnumerable<Product>**, который включает в себя экземпляры `ShoppingCart`, а также массивы объектов `Product`, как показано в [листе 4-16](#).

#### Листинг 4-16: Применение метода расширения к различным реализациям одного интерфейса

```
using LanguageFeatures.Models;
using System;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
namespace LanguageFeatures.Controllers
{
```

```

public class HomeController : Controller
{
    public string Index()
    {
        return "Navigate to a URL to show an example";
    }
    // ...другие методы действия опущены для краткости...
    public ViewResult UseExtensionEnumerable()
    {
        IEnumerable<Product> products = new ShoppingCart
        {
            Products = new List<Product> {
                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}
            }
        };
        // создание и заполнение массива объектов Product
        Product[] productArray = {
            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}
        };
        // получение общей стоимости продуктов в корзине
        decimal cartTotal = products.TotalPrices();
        decimal arrayTotal = productArray.TotalPrices();
        return View("Result",
            (object)String.Format("Cart Total: {0}, Array Total: {1}",
            cartTotal, arrayTotal));
    }
}
}

```

### **Примечание**

*Способ, которым C# массивы реализуют интерфейс `IEnumerable<T>`, немного необычен. Вы не найдете его в списке реализуемых интерфейсов в документации MSDN. Эта поддержка обрабатывается компилятором, поэтому данный код для более ранних версий C# по-прежнему компилируется. Странно, но это так. Мы могли бы использовать другой класс коллекции в этом примере, но мы хотели показать наши знания о темных углах спецификации C#. Также странно, но это так.*

---

Если вы запустите проект и нацелитесь на метод действия, вы увидите следующие результаты, которые показывают, что мы получим тот же результат от расширенного метода, независимо от того, как собраны объекты `Product`:

Cart Total: 378.40, Array Total: 378.40

## **Создание фильтрующих методов расширения**

Последнее, что мы хотим рассказать вам о методах расширения, это то, что они могут быть использованы для фильтрации объектов коллекции. Расширенный метод, который работает с `IEnumerable<T>` и который также возвращает `IEnumerable<T>`, может использовать ключевое слово `yield`, чтобы применить критерии выбора элементов в исходных данных для получения сокращенного набора результатов. В [листинге 4-17](#) показан такой метод, который мы добавили в класс `MyExtensionMethods`.

#### Листинг 4-17: Фильтрующий расширенный метод

```
using System.Collections.Generic;
namespace LanguageFeatures.Models
{
    public static class MyExtensionMethods
    {
        public static decimal TotalPrices(this IEnumerable<Product> productEnum)
        {
            decimal total = 0;
            foreach (Product prod in productEnum)
            {
                total += prod.Price;
            }
            return total;
        }
        public static IEnumerable<Product> FilterByCategory(
            this IEnumerable<Product> productEnum, string categoryParam)
        {
            foreach (Product prod in productEnum)
            {
                if (prod.Category == categoryParam)
                {
                    yield return prod;
                }
            }
        }
    }
}
```

Этот метод расширения, названный `FilterByCategory`, принимает дополнительный параметр, который позволяет нам вводить условия фильтрации, когда мы вызываем метод. Те объекты `Product`, свойство `Category` которых соответствует параметру, возвращаются в результате использования `IEnumerable<Product>`, а те, у которых не соответствует – отбрасываются. В [листе 4-18](#) показано, как используется этот метод.

#### Листинг 4-18: Использование фильтрующего расширенного метода

```
using LanguageFeatures.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
namespace LanguageFeatures.Controllers
{
    public class HomeController : Controller
    {
        public string Index()
        {
            return "Navigate to a URL to show an example";
        }
        // ... другие методы действия опущены для краткости ...
        public ViewResult UseFilterExtensionMethod()
        {
            IEnumerable<Product> products = new ShoppingCart
            {
                Products = new List<Product> {
                    new Product {Name = "Kayak", Category = "Watersports", Price = 275M},
                    new Product {Name = "Lifejacket", Category = "Watersports",
                    Price = 48.95M},
                    new Product {Name = "Soccer ball", Category = "Soccer",
                    Price = 19.50M},
                    new Product {Name = "Corner flag", Category = "Soccer",
                    Price = 34.95M}
                }
            }
        }
}
```

```
    };
    decimal total = 0;
    foreach (Product prod in products.FilterByCategory("Soccer"))
    {
        total += prod.Price;
    }
    return View("Result", (object)String.Format("Total: {0}", total));
}
}
```

Когда мы вызываем метод `FilterByCategory` для `ShoppingCart`, возвращаются только те объекты `Product`, которые находятся в категории `Soccer`. Если вы запустите проект и будете использовать метод действия `UseFilterExtensionMethod`, вы увидите следующий результат, который является общей стоимостью единиц продукции в категории `Soccer`:

Total: 54.45

## Использование лямбда-выражений

Мы можем использовать делегат, чтобы сделать наш метод `FilterByCategory` более общим. Способ, которым вызванный для каждого объекта `Product` делегат будет фильтровать объекты, как нам нужно, показан в [листинге 4-19](#). Здесь представлен расширенный метод `Filter`, который мы добавили в класс `MyExtensionMethods`.

#### Листинг 4-19: Использование делегата в расширенном методе

```
using System;
using System.Collections.Generic;
namespace LanguageFeatures.Models
{
    public static class MyExtensionMethods
    {
        public static decimal TotalPrices(this IEnumerable<Product> productEnum)
        {
            decimal total = 0;
            foreach (Product prod in productEnum)
            {
                total += prod.Price;
            }
            return total;
        }
        public static IEnumerable<Product> FilterByCategory(
            this IEnumerable<Product> productEnum, string categoryParam)
        {
            foreach (Product prod in productEnum)
            {
                if (prod.Category == categoryParam)
                {
                    yield return prod;
                }
            }
        }
        public static IEnumerable<Product> Filter(
            this IEnumerable<Product> productEnum, Func<Product, bool> selectorParam)
        {
            foreach (Product prod in productEnum)
            {
                if (selectorParam(prod))
                {
                    yield return prod;
                }
            }
        }
    }
}
```

```

        }
    }
}

```

Мы использовали `Func` в качестве фильтрующего параметра, это обозначает, что нам не нужно определять делегат в качестве типа. Делегат принимает параметр `Product` и возвращает значение `bool`, которое будет `true`, если этот объект `Product` должны быть включен в результат. Другая часть этого технического приема немного громоздкая, как показано в [листинге 4-20](#). Здесь представлены изменения, внесенные в расширенный метод `UseFilterExtensionMethod`.

#### Листинг 4-20: Использование фильтрующего расширенного метода с `Func`

```

...
public ViewResult UseFilterExtensionMethod() {
    // создаем и заполняем ShoppingCart
    IEnumerable<Product> products = new ShoppingCart {
        Products = new List<Product> {
            new Product {Name = "Kayak", Category = "Watersports", Price = 275M},
            new Product {Name = "Lifejacket", Category = "Watersports", Price = 48.95M},
            new Product {Name = "Soccer ball", Category = "Soccer", Price = 19.50M},
            new Product {Name = "Corner flag", Category = "Soccer", Price = 34.95M}
        }
    };
    Func<Product, bool> categoryFilter = delegate(Product prod) {
        return prod.Category == "Soccer";
    };
    decimal total = 0;
    foreach (Product prod in products.Filter(categoryFilter)) {
        total += prod.Price;
    }
    return View("Result", (object)String.Format("Total: {0}", total));
}
...

```

Мы продвинулись вперед в том смысле, что теперь мы можем фильтровать объекты `Product`, используя любые условия, указанные в делегате, но мы должны определять `Func` для каждого вида желаемого фильтра, а это далеко от идеала. Менее громоздкой альтернативой является использование лямбда-выражения, которое представляет собой сокращенный формат для выражения тела метода в делегате. Мы можем использовать его для замены определения нашего делегата, как показано в [листинге 4-21](#).

#### Листинг 4-21: Использование лямбда-выражения для замены определения делегата

```

...
public ViewResult UseFilterExtensionMethod() {
    // создаем и заполняем ShoppingCart
    IEnumerable<Product> products = new ShoppingCart {
        Products = new List<Product> {
            new Product {Name = "Kayak", Category = "Watersports", Price = 275M},
            new Product {Name = "Lifejacket", Category = "Watersports", Price = 48.95M},
            new Product {Name = "Soccer ball", Category = "Soccer", Price = 19.50M},
            new Product {Name = "Corner flag", Category = "Soccer", Price = 34.95M}
        }
    };
    Func<Product, bool> categoryFilter = prod => prod.Category == "Soccer";

    decimal total = 0;

    foreach (Product prod in products.Filter(categoryFilter)) {
        total += prod.Price;
    }
    return View("Result", (object)String.Format("Total: {0}", total));
}
...

```

Лямбда-выражение выделено жирным шрифтом. Параметр выражается без указания типа, который будет определен автоматически. Символы => проинсоятся как "переходит" и связывают параметр с результатом лямбда-выражения. В нашем примере параметр объекта `Product`, названный `prod`, переходит к результату `bool`, который будет верным, если `prod` параметр `Category` будет равен `Soccer`. Мы можем еще ужать синтаксис, если полностью откажемся от `Func`, как показано в [листеинге 4-22](#).

#### Листинг 4-22: Лямбда-выражение без Func

```
...
public ViewResult UseFilterExtensionMethod() {
    IEnumerable<Product> products = new ShoppingCart {
        Products = new List<Product> {
            new Product {Name = "Kayak", Category = "Watersports", Price = 275M},
            new Product {Name = "Lifejacket", Category = "Watersports", Price = 48.95M},
            new Product {Name = "Soccer ball", Category = "Soccer", Price = 19.50M},
            new Product {Name = "Corner flag", Category = "Soccer", Price = 34.95M}
        }
    };
    decimal total = 0;

    foreach (Product prod in products.Filter(prod => prod.Category == "Soccer")) {
        total += prod.Price;
    }
    return View("Result", (object)String.Format("Total: {0}", total));
}
...
```

В этом примере мы передали лямбда-выражение в качестве параметра методу `Filter`. Это хороший и естественный способ выражения фильтра, который мы хотим применить. Мы можем объединить несколько фильтров, расширяя результирующую часть лямбда-выражения, как показано в [листеинге 4-23](#).

#### Листинг 4-23: Расширение фильтрации, представленное лямбда-выражением

```
...
public ViewResult UseFilterExtensionMethod() {
    IEnumerable<Product> products = new ShoppingCart {
        Products = new List<Product> {
            new Product {Name = "Kayak", Category = "Watersports", Price = 275M},
            new Product {Name = "Lifejacket", Category = "Watersports", Price = 48.95M},
            new Product {Name = "Soccer ball", Category = "Soccer", Price = 19.50M},
            new Product {Name = "Corner flag", Category = "Soccer", Price = 34.95M}
        }
    };
    decimal total = 0;
    foreach (Product prod in products.Filter(prod => prod.Category == "Soccer" || prod.Price > 20))
    {
        total += prod.Price;
    }
    return View("Result", (object)String.Format("Total: {0}", total));
}
...
```

Это лямбда-выражение будет соответствовать объектам `Product`, которые находятся в категории `Soccer` или чье свойство `Price` больше 20.

## Другие формы лямбда-выражений

Нам не обязательно выражать логику делегатов в лямбда-выражении. Мы можем просто вызвать метод, вот так:

```
prod => EvaluateProduct(prod)
```

Если нам нужно лямбда-выражение для делегата, который имеет несколько параметров, мы должны заключить параметры в круглые скобки, например, вот так:

```
(prod, count) => prod.Price > 20 && count > 0
```

И, наконец, если нам в лямбда-выражении нужна логика, требующая более одного выражения, мы реализуем ее, используя фигурные скобки ({ }), и завершаем данный кусок кода при помощи `return`, вот так:

```
(prod, count) => {
    //...несколько выражений кода
    return result;
}
```

Вам не обязательно использовать лямбда-выражения в коде, но они являются аккуратным способом выражения сложных функций, которые становятся простыми в понимании и вполне читабельными. Мы их очень любим, и вы увидите, что в этой книге они часто будут использоваться.

## Автоматическое определение типа

Ключевое слово `var` в C# позволяет определять локальные переменные без явного указания типа этих переменных, как показано в [листе 4-24](#). Это называется *автоматическим определением типа* или *неявной типизацией*.

**Листинг 4-24:** Использование автоматического определения типа

```
...
var myVariable = new Product { Name = "Kayak", Category = "Watersports", Price = 275M };
string name = myVariable.Name; // правильно
int count = myVariable.Count; // ошибка компилятора
...
```

Нельзя сказать, что `myVariable` не имеет типа. Это мы просим компилятор, чтобы он обрабатывал ее в коде. По следующим выражениям видно, что компилятор будет работать только с членами выбранного класса, в данном случае с объектами `Product`.

## Использование анонимных типов

Объединив инициализаторы объектов и автоматическое определение типа, мы можем создавать простые объекты для хранения данных без необходимости указания соответствующего класса или структуры. В [листе 4-25](#) представлен пример.

**Листинг 4-25:** Создание анонимного типа

```
...
var myAnonType = new {
    Name = "MVC",
```

```
Category = "Pattern"  
};  
...
```

В этом примере `myAnonType` – это анонимно типизированный объект. Это не обозначает, что он является динамическим, как например, JavaScript переменные, которые являются динамически типизированными (слабо типизированными). Это просто обозначает, что тип будет автоматически определен компилятором. Строгая типизация по-прежнему соблюдается. Например, вы можете получать и устанавливать только те свойства, которые были определены в инициализаторе.

C# компилятор генерирует класс, основываясь на имени и типе параметров в инициализаторе. Два анонимно типизированных объекта, которые имеют одинаковые имена свойств и типы, будут относиться к одному и тому же автоматически созданному классу. Это обозначает, что мы можем создать массивы анонимно типизированных объектов, как показано в [листинге 4-26](#). Здесь представлен метод действия `CreateAnonArray`, который мы добавили в контроллер `Home`.

#### Листинг 4-26: Создание массива анонимно типизированных объектов

```
using LanguageFeatures.Models;  
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Web;  
using System.Web.Mvc;  
namespace LanguageFeatures.Controllers  
{  
    public class HomeController : Controller  
    {  
        public string Index()  
        {  
            return "Navigate to a URL to show an example";  
        }  
        // ...другие методы действия опущены для краткости ...  
        public ViewResult CreateAnonArray()  
        {  
            var oddsAndEnds = new[] {  
                new { Name = "MVC", Category = "Pattern" },  
                new { Name = "Hat", Category = "Clothing" },  
                new { Name = "Apple", Category = "Fruit" }  
            };  
            StringBuilder result = new StringBuilder();  
            foreach (var item in oddsAndEnds)  
            {  
                result.Append(item.Name).Append(" ");  
            }  
            return View("Result", (object)result.ToString());  
        }  
    }  
}
```

Обратите внимание, что при объявлении массива мы используем `var`. Мы должны это сделать, потому что у нас нет типа, который мы могли бы указать, как в типизированном массиве. И хотя мы не определили класс ни для одного из этих объектов, мы все еще можем перечислить содержимое массива и прочитать значение свойства `Name` каждого из них. Это важно, потому что без этой возможности мы вообще не смогли бы создавать массивы анонимно типизированных объектов. Или, вернее, мы могли бы создавать такие массивы, но мы не могли бы ничего полезного с ними делать. Если вы запустите этот пример и будете работать с данным методом действия, вы увидите следующий результат:

MVC Hat Apple

# Использование LINQ

Все возможности, которые мы описывали до сих пор, хорошо работают с LINQ. Мы любим LINQ. Это прекрасное и важное дополнение к .NET. Если вы никогда не использовали LINQ, вы очень многое упустили. LINQ представляет собой SQL-подобный синтаксис для выборки данных в классах. Представьте себе, что у нас есть набор объектов `Product`, и мы хотим получить три из них с самыми высокими ценами и передать их методу `View`. Без LINQ мы в итоге получили бы нечто похожее на [листинг 4-27](#), который показывает метод действия `FindProducts`, добавленный в контроллер `Home`.

## Листинг 4-27: Выборка без LINQ

```
public ViewResult FindProducts()
{
    Product[] products = {
        new Product {Name = "Kayak", Category = "Watersports", Price = 275M},
        new Product {Name = "Lifejacket", Category = "Watersports", Price = 48.95M},
        new Product {Name = "Soccer ball", Category = "Soccer", Price = 19.50M},
        new Product {Name = "Corner flag", Category = "Soccer", Price = 34.95M}
    };
    // определяем массив для результатов
    Product[] foundProducts = new Product[3];
    // сортируем содержание массива
    Array.Sort(products, (item1, item2) =>
    {
        return Comparer<decimal>.Default.Compare(item1.Price, item2.Price);
    });
    // получаем три первых элемента массива в качестве результата
    Array.Copy(products, foundProducts, 3);
    // создаем результат
    StringBuilder result = new StringBuilder();
    foreach (Product p in foundProducts)
    {
        result.AppendFormat("Price: {0} ", p.Price);
    }
    return View("Result", (object)result.ToString());
}
```

С LINQ мы можем значительно упростить процесс выборки, как показано в [листинге 4-28](#).

## Листинг 4-28: Использование LINQ для выборки данных

```
public ViewResult FindProducts()
{
    Product[] products = {
        new Product {Name = "Kayak", Category = "Watersports", Price = 275M},
        new Product {Name = "Lifejacket", Category = "Watersports", Price = 48.95M},
        new Product {Name = "Soccer ball", Category = "Soccer", Price = 19.50M},
        new Product {Name = "Corner flag", Category = "Soccer", Price = 34.95M}
    };
    var foundProducts = from match in products
                        orderby match.Price descending
                        select new
                        {
                            match.Name,
                            match.Price
                        };
    // создаем результат
    int count = 0;
    StringBuilder result = new StringBuilder();
    foreach (var p in foundProducts)
    {
```

```

        result.AppendFormat("Price: {0} ", p.Price);
        if (++count == 3)
        {
            break;
        }
    }
    return View("Result", (object)result.ToString());
}
}

```

Это намного аккуратнее. Вы можете увидеть SQL-подобную выборку, выделенную жирным шрифтом. Мы располагаем объекты `Product` в порядке убывания по свойству `Price` и используем ключевое слово `select`, чтобы вернуть анонимный тип, содержащий только свойства `Name` и `Price`. Этот стиль LINQ известен как *синтаксис запросов*, и это тот стиль, который разработчики считают наиболее удобным, когда они начинают использовать LINQ. Такая выборка возвращает один анонимно типизированный объект для каждого `Product` в массиве, который мы использовали в исходном запросе, поэтому нам нужно поработать с результатами, чтобы получить первые три объекта и вывести их на экран.

Если же мы готовы отказаться от простоты синтаксиса запросов, мы сможем ощутить намного больше мощи LINQ. Альтернативой является *синтаксис точечной нотации*, или *точечная нотация*, которая основана на методах расширения. В [листинге 4-29](#) показано, как мы можем использовать этот альтернативный синтаксис для обработки наших объектов `Product`.

#### **Листинг 4-29:** Использование точечной нотации LINQ

```

public ViewResult FindProducts() {
    Product[] products = {
        new Product {Name = "Kayak", Category = "Watersports", Price = 275M},
        new Product {Name = "Lifejacket", Category = "Watersports", Price = 48.95M},
        new Product {Name = "Soccer ball", Category = "Soccer", Price = 19.50M},
        new Product {Name = "Corner flag", Category = "Soccer", Price = 34.95M}
    };
    var foundProducts = products.OrderByDescending(e => e.Price)
        .Take(3)
        .Select(e => new {
            e.Name,
            e.Price
        });
    StringBuilder result = new StringBuilder();
    foreach (var p in foundProducts) {
        result.AppendFormat("Price: {0} ", p.Price);
    }
    return View("Result", (object)result.ToString());
}

```

Мы будем первыми, кто признает, что на эту LINQ выборку, выделенную жирным шрифтом, не так приятно смотреть, как на синтаксис запроса. Однако, не для всех LINQ функций имеются соответствующие ключевые слова C#. Как серьезные LINQ программисты мы должны перейти к использованию методов расширения. Каждый из методов расширение LINQ в листинге применяется к `IEnumerable<T>` и возвращает тоже `IEnumerable<T>`, что позволяет связывать методы цепочкой, чтобы формировать сложные выборки.

#### **Примечание**

*Все методы расширения LINQ находятся в пространстве имен `System.Linq`, которое вы должны вставить в код при помощи ключевого слова `using`, прежде чем делать выборку. Visual Studio автоматически добавляет пространство имен `System.Linq` к классам контроллера, но вы можете добавить его вручную в любом другом месте MVC проекта.*

Метод `OrderByDescending` сортирует элементы в исходных данных. В этом случае лямбда-выражение возвращает значение, которое мы хотим использовать для сравнения. Метод `Take` возвращает указанное число элементов с начала цепочки результатов (это то, что мы не могли сделать, используя синтаксис запроса). Метод `Select` позволяет нам проектировать нужные результаты. В данном случае мы проектируем анонимный объект, который содержит свойства `Name` и `Price`. Обратите внимание, что нам даже не нужно указывать имена свойств в анонимном типе. C# сделал вывод, основываясь на свойствах, которые мы вставили в метод `Select`.

В [таблице 4-1](#) описаны наиболее полезные методы расширения LINQ. Мы много используем LINQ в этой книге, и, возможно, вы захотите вернуться к этой таблице, когда повстречаете метод расширения, с которым вы не сталкивались раньше. Все методы LINQ, представленные в таблице, работают с `IEnumerable<T>`.

**Таблица 4-1:** Некоторые полезные методы расширения LINQ

Метод расширения	Описание	Отложенный
<code>All</code>	Возвращает <code>true</code> , если все элементы в исходных данных соответствуют утверждению	Нет
<code>Any</code>	Возвращает <code>true</code> , если, как минимум, один из элементов в исходных данных соответствуют утверждению	Нет
<code>Contains</code>	Возвращает <code>true</code> , если исходные данные содержат указанный элемент или значение	Нет
<code>Count</code>	Возвращает число элементов в исходных данных	Нет
<code>First</code>	Возвращает первый элемент из исходных данных	Нет
<code>FirstOrDefault</code>	Возвращает первый элемент из исходных данных или значение по умолчанию, если элементов нет	Нет
<code>Last</code>	Возвращает последний элемент из исходных данных	Нет
<code>LastOrDefault</code>	Возвращает последний элемент из исходных данных или значение по умолчанию, если элементов нет	Нет
<code>Max, Min</code>	Возвращает самое большое или самое маленькое значение, указанное лямбда-выражением	Нет
<code>OrderBy, OrderByDescending</code>	Сортирует исходные данные, основываясь на значении, возвращенном лямбда-выражением	Да
<code>Reverse</code>	Меняет порядок элементов в исходных данных	Да
<code>Select</code>	Проектирует результат выборки	Да
<code>SelectMany</code>	Проектирует каждый элемент данных в последовательность элементов, а затем объединяет все эти результирующие последовательности в одну последовательность	Да

Метод расширения	Описание	Отложенный
Single	Возвращает первый элемент из исходных данных или выбрасывает исключение, если есть несколько совпадений	Нет
SingleOrDefault	Возвращает первый элемент из исходных данных или значение по умолчанию, если элементов нет, или выбрасывает исключение, если есть несколько совпадений	Нет
Skip, SkipWhile	Пропускает указанное число элементов или пропускает элементы, соответствующие утверждению	Да
Sum	Подсчитывает выбранные значения	Нет
Take, TakeWhile	Выбирает указанное число элементов от начала исходных данных или выбирает элементы, пока идет соответствие утверждению	Да
ToArray, ToDictionary, ToList	Конвертирует исходные данные в массив или другие типы коллекций	Нет
Where	Фильтрует элементы из исходных данных, которые не соответствуют утверждению	Да

## Отложенные выборки LINQ

Вы заметили, что в [таблице 4-1](#) содержится столбец «Отложенный». Есть интересный момент в том, как методы расширения выполняются в LINQ запросе. Выборка, которая содержит только отложенные методы, не будет выполняться до тех пор, пока не будут перечислены элементы результата, как показано в [листинге 4-30](#). Здесь представлено простое изменение в методе действия `FindProducts`.

**Листинг 4-30:** Использование в выборке отложенных методов расширения LINQ

```
public ViewResult FindProducts()
{
    Product[] products = {
        new Product {Name = "Kayak", Category = "Watersports", Price = 275M},
        new Product {Name = "Lifejacket", Category = "Watersports", Price = 48.95M},
        new Product {Name = "Soccer ball", Category = "Soccer", Price = 19.50M},
        new Product {Name = "Corner flag", Category = "Soccer", Price = 34.95M}
    };
    var foundProducts = products.OrderByDescending(e => e.Price)
        .Take(3)
        .Select(e => new
    {
        e.Name,
        e.Price
    });
    products[2] = new Product { Name = "Stadium", Price = 79600M };
    StringBuilder result = new StringBuilder();
    foreach (var p in foundProducts)
    {
        result.AppendFormat("Price: {0} ", p.Price);
    }
    return View("Result", (object)result.ToString());
}
```

Между определением LINQ выборки и перечислением результатов мы изменили один из элементов массива products. Результат этого примера выглядит следующим образом:

```
Price: 79600 Price: 275 Price: 48.95
```

Вы видите, что запрос не обрабатывается до получения результатов перечисления, и поэтому изменение, которое мы сделали – введение Stadium в массив Product – отображается в выходных данных.

#### Совет

Одна интересная особенность отложенных методов расширения LINQ заключается в том, что запросы обрабатываются с нуля каждый раз после перечисления результатов, это обозначает, что вы можете выполнить выборку повторно в качестве исходных данных для изменений.

В отличие от этого, использование любого из не отложенных (nondeferred) методов расширения приводит к тому, что выборка LINQ выполняется немедленно. В [листинге 4-31](#) показан метод действия SumProducts, которые мы добавили в контроллер Home.

#### Листинг 4-31: Немедленно выполняемая выборка LINQ

```
public ViewResult SumProducts()
{
    Product[] products = {
        new Product {Name = "Kayak", Category = "Watersports", Price = 275M},
        new Product {Name = "Lifejacket", Category = "Watersports", Price = 48.95M},
        new Product {Name = "Soccer ball", Category = "Soccer", Price = 19.50M},
        new Product {Name = "Corner flag", Category = "Soccer", Price = 34.95M}
    };

    var results = products.Sum(e => e.Price);

    products[2] = new Product { Name = "Stadium", Price = 79500M };

    return View("Result", (object)String.Format("Sum: {0:c}", results));
}
```

В этом примере используется метод Sum, который является не отложенным и выдает следующий результат:

```
Sum: $378.40
```

Вы видите, что элемент Stadium с его гораздо более высокой ценой не был включен в результаты. Это произошло потому, что метод Sum сразу же выдает результат, как только вызывается этот метод, поскольку он является не отложенным методом.

#### LINQ и интерфейс IQueryble<T>

Можно повстречать различные вариации LINQ, хотя используется он всегда практически одинаково. Одной из разновидностей является LINQ to Objects, и эту разновидность мы использовали в примерах данной главы. LINQ to Objects позволяет делать выборку C# объектов, которые находятся в памяти. Другая разновидность, LINQ to XML – это очень удобный и мощный способ создания, обработки и выборки XML содержания. Параллельный LINQ (Parallel LINQ, PLINQ) является расширенной версией LINQ to Object, которая поддерживает выполнение LINQ выборок одновременно на нескольких процессорах или ядрах.

Особый интерес для нас представляет LINQ to Entities, который позволяет делать LINQ запросы к данным, полученным из Entity Framework. Entity Framework является ORM фреймворком Microsoft, представляющий собой часть более широкой платформы ADO.NET. ORM позволяет работать с реляционными данными при помощи C# объектов. И это тот механизм, который мы будем использовать в данной книге для доступа к данным, хранящимся в базах данных. Вы увидите, как используются Entity Framework и LINQ to Entities, в главе 4. Но мы также хотели упомянуть интерфейс `IQueryable<T>` во время представления LINQ.

Интерфейс `IQueryable<T>` является производным от `IEnumerable<T>` и используется для обозначения результата выборки, выполненной в отношении конкретного источника данных. В наших примерах это будет база данных SQL Server. Вообще, нет необходимости использовать `IQueryable<T>` напрямую. Одна из приятных особенностей LINQ заключается в том, что одна и та же выборка может быть выполнена для нескольких типов исходных данных (объектов, XML, баз данных и т. д.). Когда вы увидите, что мы используем `IQueryable<T>` в примерах дальнейших глав, то вы поймете, что мы делаем это, чтобы было ясно, что мы имеем дело с данными, которые приходят из базы данных.

## Использование асинхронных методов

Одним из самых серьезных дополнений C# в .NET 4.5 является улучшенный способ работы *асинхронных методов*. Асинхронные методы скрыты и работают в фоновом режиме, а затем они уведомляют вас, когда завершают свою работу, что позволяет коду выполнять другие задачи, пока идет работа в фоновом режиме. Асинхронные методы являются важным инструментом в устранении узких мест в коде и позволяют приложениям работать параллельно на нескольких процессорах и процессорных ядрах.

C# и .NET отлично поддерживают асинхронные методы, но тогда код становится немного громоздким, и разработчики, которые не привыкли к параллельному программированию, часто вязнут в необычном синтаксисе. В качестве простого примера в [листе 4-32](#) представлен асинхронный метод, который называется `GetPageLength`. Мы определили его в классе `MyAsyncMethods` и добавили в папку `Models`.

**Листинг 4-32:** Простой асинхронный метод

```
using System.Net.Http;
using System.Threading.Tasks;
namespace LanguageFeatures.Models
{
    public class MyAsyncMethods
    {
        public static Task<long?> GetPageLength()
        {
            HttpClient client = new HttpClient();
            var httpTask = client.GetAsync("http://apress.com");
            // мы можем здесь делать другие вещи, пока мы ждем
            // окончания HTTP запроса
            return httpTask.ContinueWith((Task<HttpResponseMessage> antecedent) =>
            {
                return antecedent.Result.Content.Headers.ContentLength;
            });
        }
    }
}
```

Это простой метод, который использует объект `System.Net.Http.HttpClient` для запроса содержимого страницы Apress и возвращает ее длину. Мы выделили ту часть метода, которая может привести к путанице, что является примером *возобновления задачи*.

.NET будет работать асинхронно используя Task. Объекты Task строго типизированы и основываются на результате, который возвращает работа в фоновом режиме. Так что когда мы вызываем метод HttpClient.GetAsync, то, что нам вернется, будет Task<HttpResponseMessage>. Это говорит нам о том, что запрос будет выполняться в фоновом режиме и о том, что результат запроса будет объектом HttpResponseMessage.

### *Совет*

*Когда мы используем такие слова, как фоновый режим, мы опускаем много деталей, чтобы сосредоточиться на ключевых моментах, которые важны для MVC .NET поддержка асинхронных методов и параллельного программирования в целом превосходна, и мы советуем вам узнать как можно больше об этом, если вы хотите создавать по-настоящему высокопроизводительные приложения, которые могут использовать преимущества многоядерных и многопроцессорных аппаратных средств. Мы вернемся к асинхронным методам MVC в главе 17.*

---

Та часть, в которой могут увязнуть большинство программистов, это возобновление задачи. Это механизм, при помощи которого вы указываете, что должно произойти, когда будет выполнена фоновая задача. В этом примере мы использовали метод ContinueWith для обработки объекта HttpResponseMessage который мы получаем от метода HttpClient.GetAsync. Это мы сделали при помощи лямбда-выражения, которое возвращает значение свойства, возвращающего длину контента, которой мы получили от веб-сервера Apress. Обратите внимание, что мы дважды используем ключевое слово return:

```
return httpTask.ContinueWith((Task<HttpResponseMessage> antecedent) => {
    return antecedent.Result.Content.Headers.ContentLength;
});
```

Это та часть, которая доставляет головную боль. Первое использование ключевого слова return указывает на то, что мы возвращаем объект Task<HttpResponseMessage>, который после выполнения задачи вернет (будет использовать return) длину заголовка ContentLength. Заголовок ContentLength возвращает результат long?, и это обозначает, что результат нашего метода GetPageLength будет Task<long?>:

```
public static Task<long?> GetPageLength() {
```

Не волнуйтесь, если вам это покажется бессмысленным: вы в этом не одиноки, и это очень простой пример. Сложные асинхронные операции могут связывать большое число задач, используя метод ContinueWith, который создает трудный для чтения и сложный в поддержке код.

## Применение ключевых слов async и await

Microsoft представил два новых ключевых слова в C#, которые специально предназначены для упрощения использования асинхронных методов, таких как HttpClient.GetAsync. Новые ключевые слова – это async и await, и вы можете увидеть в [листинге 4-33](#), как мы использовали их для упрощения нашего примера.

### Листинг 4-33: Использование ключевых слов async и await

```
using System.Net.Http;
using System.Threading.Tasks;
namespace LanguageFeatures.Models
{
    public class MyAsyncMethods
    {
```

```
public async static Task<long?> GetPageLength()
{
    HttpClient client = new HttpClient();
    var httpMessage = await client.GetAsync("http://apress.com");
    // мы можем здесь делать другие вещи, пока мы ждем
    // окончания HTTP запроса
    return httpMessage.Content.Headers.ContentLength;
}
```

Мы использовали ключевое слово `await` при вызове асинхронного метода. Это говорит компилятору C#, что мы хотим подождать результат `Task`, который возвращает метод `GetAsync`, а затем продолжить выполнение других операторов в том же методе.

Применение ключевого слова `await` обозначает, что мы можем обрабатывать результат, полученный из метода `GetAsync`, как будто это обычный метод, и просто присвоить объект `HttpResponseMessage`, который он возвращает, переменной. И что еще лучше, мы можем использовать ключевое слово `return` в обычном порядке для получения результата от другого метода, в данном случае, значение свойства `ContentLength`. Это гораздо более естественный способ, и это обозначает, что мы не должны беспокоиться о методе `ContinueWith` и многократном использовании ключевого слова `return`.

При использовании ключевого слова `await` необходимо также добавлять ключевое слово `async` в описание метода, как мы это сделали в нашем примере. И тип, возвращенный методом, не изменится: наш метод из примера `GetPageLength` по-прежнему возвращает `Task<long?>`. Это происходит потому, что `await` и `async` реализованы при помощи некоторых классных уловок компилятора. То есть, они позволяют нам использовать более естественный синтаксис, но они не меняют то, что происходит в методах, к которым они применяются. Кто-то, кто вызывает наш метод `GetPageLength`, по-прежнему имеет дело с результатом `Task<long?>`, потому что существует фоновое действие, результатом которого является значение `long?`. Хотя, конечно, программисты сами выбирают, использовать ли им `await` и `async`.

## Примечание

*Возможно, вы заметили, что мы не предоставили MVC примера для тестирования ключевых слов `async` и `await`. Это потому что использование асинхронных методов в MVC контроллерах требует специальных технических навыков. И до главы 17 мы дадим вам еще много полезной информации по этой теме.*

## Резюме

В этой главе мы начали с обзора ключевых функций языка C#, о которых должны знать эффективные MVC программисты. Эти функции объединены в LINQ, который мы будем использовать для выборки данных в этой книге. Как мы уже говорили, мы большие поклонники LINQ, и он играет важную роль в MVC приложениях. Мы также показали вам ключевые слова `async` и `await`, которые облегчают работу с асинхронными методами: это тема, к которой мы вернемся в главе 17, когда мы покажем вам передовые технические приемы для интеграции асинхронного программирования с вашими MVC контроллерами.

В следующей главе мы обратим ваше внимание на движок представления Razor, который является механизмом для вставки динамических данных в представления.

# Работа с Razor

Razor – это движок представления, который Microsoft представил в MVC 3 и который был немного переделан в MVC 4 (хотя изменения являются относительно незначительными). Движок представления обрабатывает ASP.NET контент и ищет инструкции, как правило, для вставки динамического контента в выходные данные, отправленные браузеру. Microsoft поддерживает два вида движков: движок ASPX работает с тегами <% и %>, которые являлись основой развития ASP.NET в течение многих лет. А движок Razor, который работает с отдельными областями контента, обозначается символом @.

По большому счету, если вы знакомы с синтаксисом <% %>, Razor не доставит слишком много проблем, хотя и есть несколько новых правил. В этом разделе мы дадим вам краткий обзор синтаксиса Razor, так что вы сможете распознать новые элементы, когда вы их встретите. Мы не собираемся углубленно изучать Razor в этой главе, скорее это ускоренный курс синтаксиса. Более подробно мы изучим Razor далее в этой книге.

## Совет

Razor тесно связаны с MVC, но с появлением ASP.NET 4.5 движок представления Razor также поддерживает ASP.NET Web Pages.

## Создание проекта примера

Для демонстрации возможностей и синтаксиса Razor мы создали новый проект Visual Studio, используя шаблон ASP.NET MVC 4 Web Application и выбрав вариант Empty.

### Определение модели

Мы будем использовать очень простую доменную модель и воспроизведем тот же доменный класс Product, который мы использовали в первой части книги. Добавьте файл класса Product.cs в папку Models и убедитесь, что содержание соответствует тому, что показано в [листинге 5-1](#).

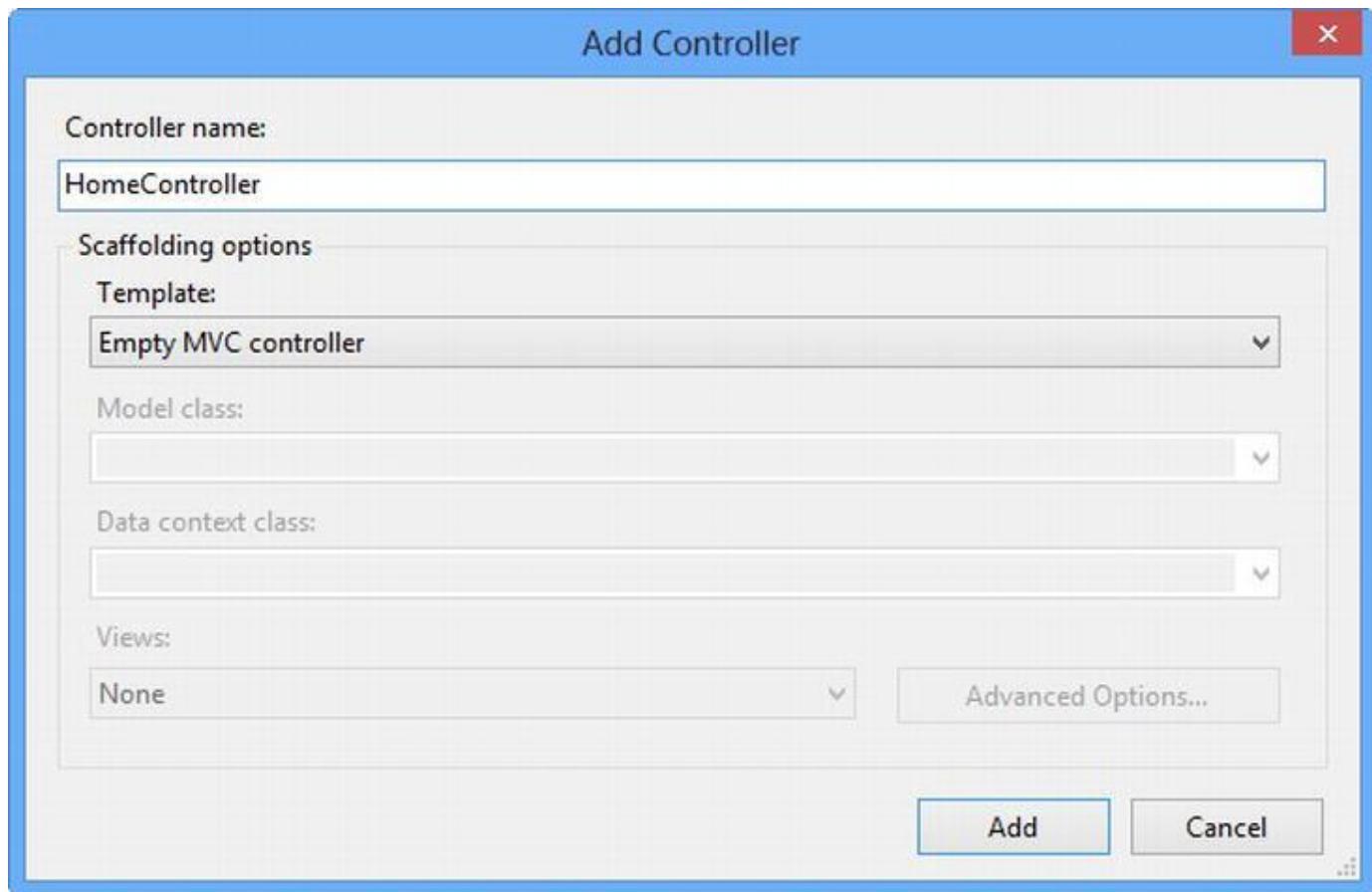
#### Листинг 5-1: Создание простого класса доменной модели

```
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 контроллер в проект, щелкните правой кнопкой мыши в вашем проекте по папке Controllers и выберите Add, а затем Controller, из всплывающего меню. Назовите его HomeController и выберите Empty MVC Controller в опции Template, как показано на [рисунке 5-1](#).

**Рисунок 5-1:** Создание ProductController



Нажмите кнопку Add, чтобы создать класс Controller, а затем измените содержимое файла, чтобы оно соответствовало [листингу 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 `Index`, в котором мы создаем и заполняем свойства объекта `Product`. Мы передаем `Product` методу `View`, поэтому он используется в качестве модели, когда

воспроизводится представление. Мы не указываем имя файла представления, когда вызываем метод View, поэтому для метода действия будет использоваться представление по умолчанию (мы создадим файл представления следующим).

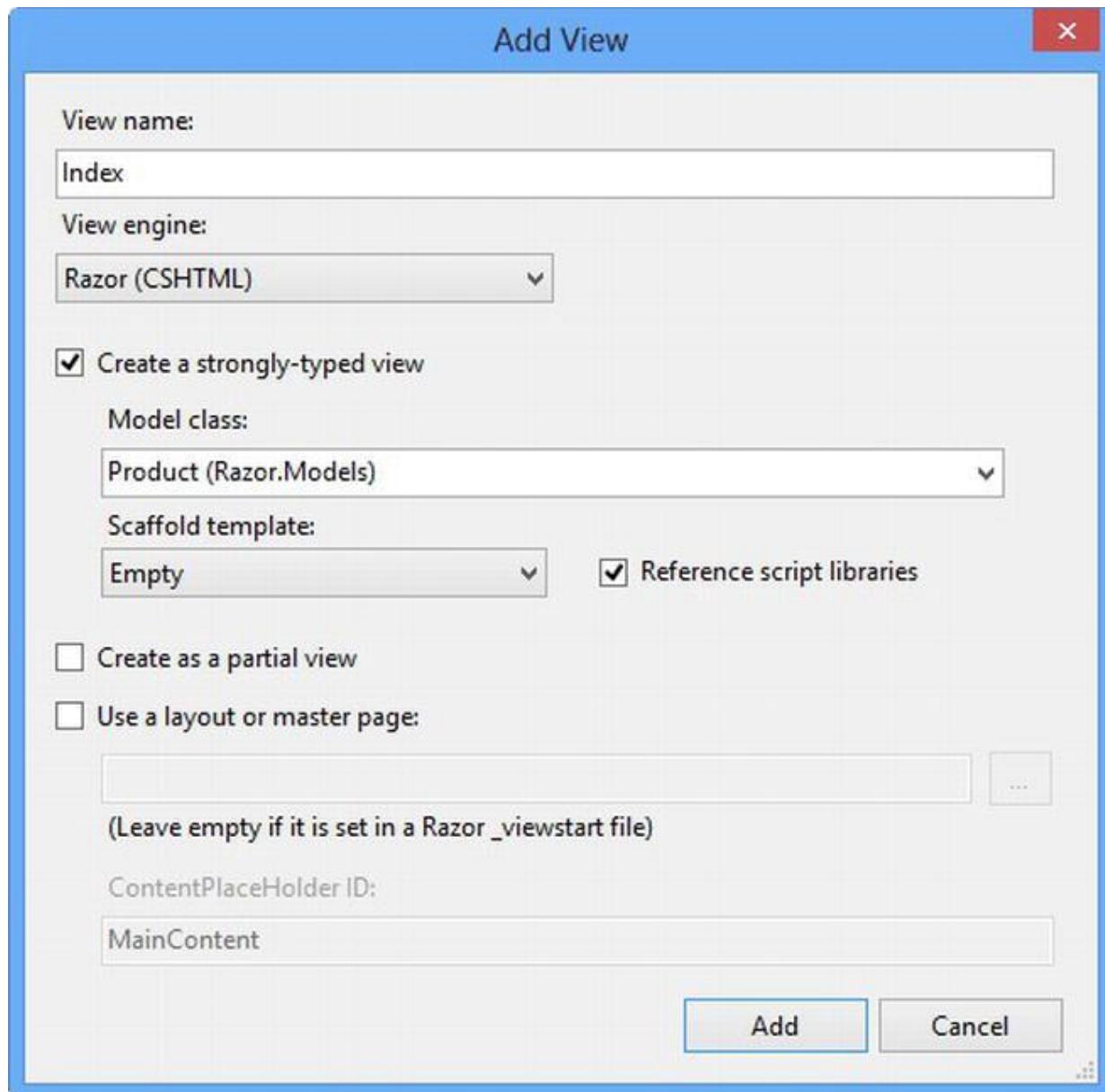
## Создание представления

Чтобы создать представление, щелкните правой кнопкой мыши по методу Index класса HomeController и выберите Add View. Проверьте опцию, что создается строго типизированное представление и выберите класс из раскрывающегося списка Product, как показано на [рисунке 5-2](#).

Примечание

Если вы не видите в раскрывающемся списке класс Product, скомпилируйте ваш проект и попробуйте создать представление еще раз. Visual Studio не распознает классы модели, пока они не скомпилированы.

Рисунок 5-2: Добавление представления



Убедитесь, что отключена опция для использования макета и мастер страницы (use a layout or master page), как показано на рисунке. Нажмите кнопку Add, чтобы создать представление, которое появится в папке Views/Home и будет называться Index.cshtml. Файл представления будет открыт для редактирования, и вы увидите, что это тот же базовый файл представления, который мы создавали в предыдущей главе, как показано в [листеинге 5-3](#).

#### Листинг 5-3: Простое Razor представление

```
@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 следующих разделах мы рассмотрим различные аспекты Razor представления и покажем некоторые вещи, которые вы можете сделать с ним.

При изучении Razor важно понимать, что представления существуют, чтобы выразить одну или несколько частей модели пользователю: а это обозначает генерирование HTML, который отображает данные, полученные от одного или нескольких объектов. Если вы запомните, что мы всегда пытаемся выстроить HTML страницу, которая может быть отправлена клиенту, то все, что делает Razor, будет иметь для вас смысл.

#### Примечание

Мы повторим некоторую информацию, которой мы коснулись в главе 2, в следующих разделах. Мы хотим собрать все в одном месте, куда вы можете посмотреть, если вам нужно будет найти функцию Razor, и мы думали, что небольшое количество дублирования этого стоит.

## Работа с объектом модели

Давайте начнем с первой строки в представлении:

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

Операторы Razor начинаются с символа @. В данном случае выражение @model объявляет тип объекта модели, который мы передадим в представление из метода действия. Это позволяет нам обратиться к методам, полям и свойствам объекта модели представления через @Model, как показано в [листеинге 5-4](#). Здесь представлено простое дополнение к методу Index.

#### Листинг 5-4: Обращение к свойству объекта модели в Razor представлении

```
@model Razor.Models.Product
{
    Layout = null;
```

```
}

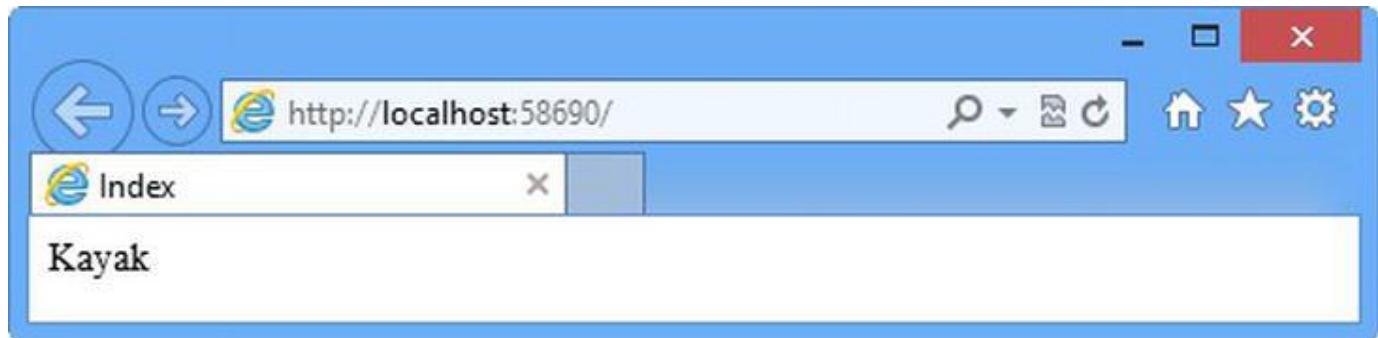
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
        @Model.Name
    </div>
</body>
</html>
```

#### Примечание

*Обратите внимание, что мы объявили тип объекта модели представления при помощи @model (маленькая m), а получили доступ к свойству Name при помощи @Model (заглавная M). Это кажется немного запутанным, когда вы начинаете работать с Razor, но вы очень быстро к этому привыкнете.*

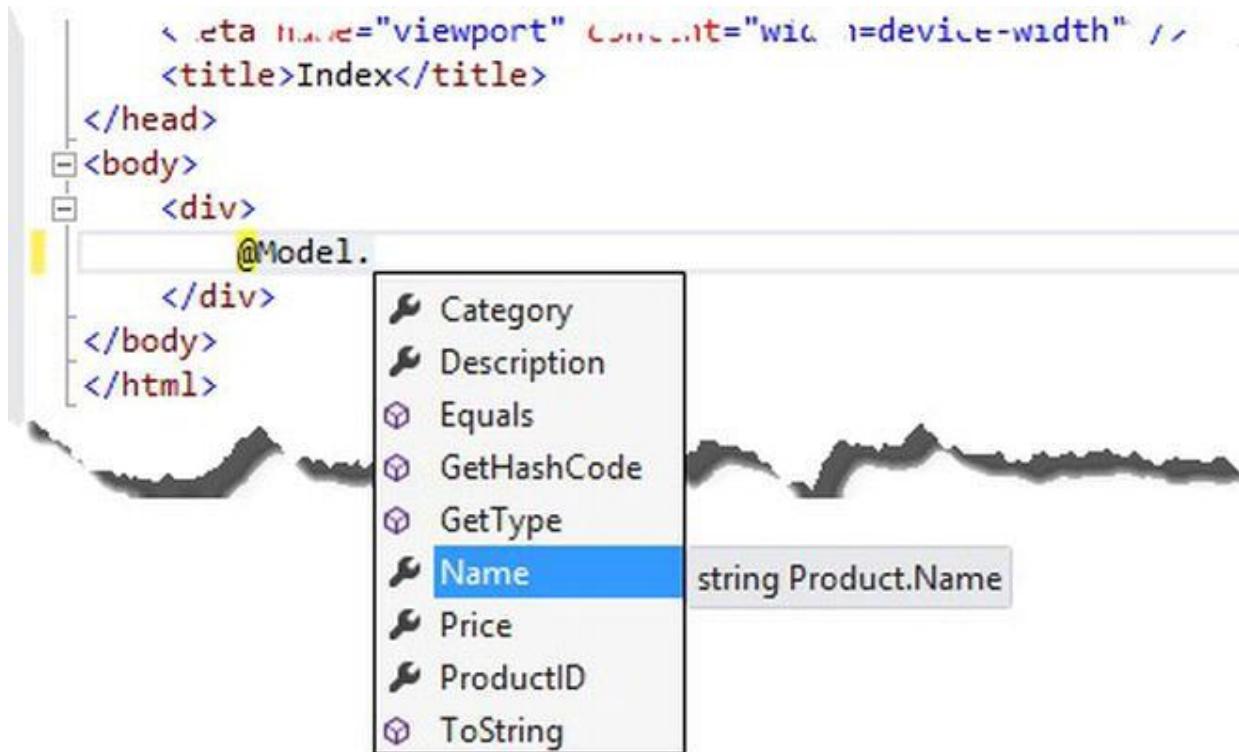
Если вы запустите проект, вы увидите результат, показанный на [рисунке 5-3](#). Вам не нужно указывать конкретный URL, потому что по умолчанию в MVC проекте запрос для корневого URL (/) направляется к методу действия Index в контроллере Home, хотя мы покажем вам, как это изменить, в главе 13.

**Рисунок 5-3:** Результат прочтения значения свойства в представлении



При помощи выражения @model мы говорим MVC, с каким видом объекта мы будем работать, а Visual Studio использует это в несколькими способами. Во-первых, когда вы пишете код представления, Visual Studio покажет вам несколько вариантов, касательных имен членов объекта, если вы наберете @Model и поставите точку, как показано на [рисунке 5-4](#). Это очень похоже на то, как работает автозаполнение для лямбда-выражений, которые передаются вспомогательным методам HTML, о чем мы рассказывали в [главе 4](#).

Рисунок 5-4: Visual Studio предлагает имена для членов объекта, основываясь на выражении @Model



Не менее полезным является то, что Visual Studio будет отмечать ошибки, если есть проблемы с членами объекта модели представления, к которым вы обращаетесь. Вы можете увидеть пример этого на [рисунке 5-5](#), где мы попытались обратиться к методу @Model.NotARealProperty. Visual Studio понял, что у класса Product нет такого свойства, и выделил в редакторе ошибку.

Рисунок 5-5: Visual Studio обозначает проблему с выражением, содержащим @Model



## Работа с макетами

Другое Razor выражение в файле представления Index.cshtml следующее:

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

Это пример блока кода Razor, который позволяет нам включать в представление C# выражения. Блок кода открывается @ { и закрывается }, а выражения, которые он содержит, обрабатываются тогда, когда отрабатывает представление.

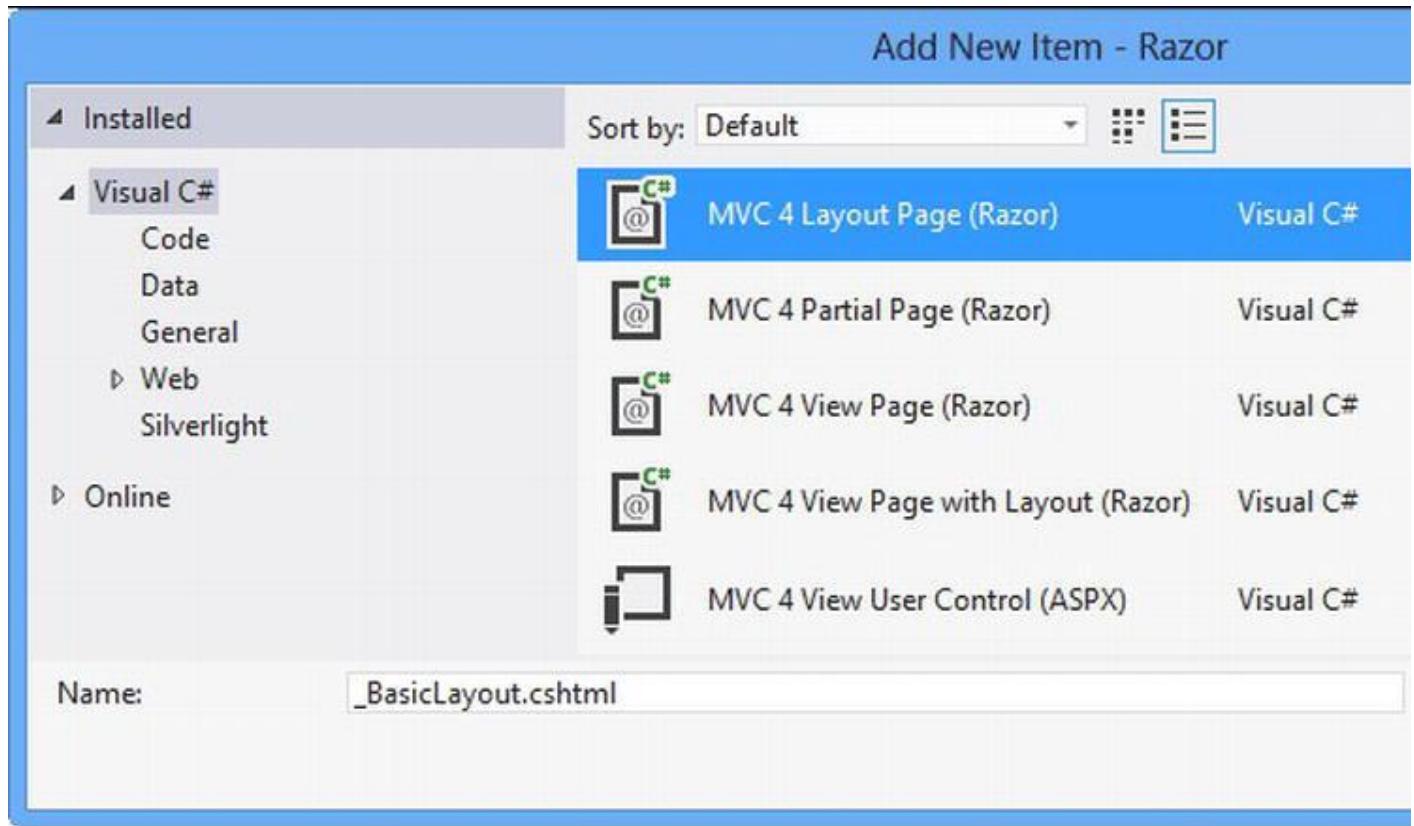
Конкретно этот блок кода устанавливает значение свойства Layout на null. Как мы объясним в главе 18, в MVC приложении представления Razor компилируются в C# классы, и используемый базовый класс определяет свойство Layout. Мы покажем вам, как все это работает, в главе 18, но результат установки свойства Layout на null заключается в том, что мы говорим MVC фреймоврку, что наше представление является автономным, и оно будет показывать все содержимое, которое мы должны вернуть клиенту.

Автономные представления хороши для простых приложений-примеров, но реальный проект может иметь множество представлений, и макеты являются эффективными шаблонами, которые содержат разметку, используемую для создания логичности и постоянства в веб приложении. Это может заключаться в том, что в приложение будут включены необходимые JavaScript библиотеки, или в создании общего гармоничного вида всего приложения.

## Создание макета

Для создания макета щелкните правой кнопкой мыши по папке Views в Solution Explorer, нажмите на Add New Item в меню Add и выберите шаблон MVC 4 Layout Page (Razor), как показано на [рисунке 5-6](#).

Рисунок 5-6: Создание нового макета



Назовите файл \_BasicLayout.cshtml и нажмите кнопку Add, чтобы создать файл. В листинге 5-5 показано содержимое файла, созданного Visual Studio.

## Примечание

Файлы в папке Views, чьи имена начинаются с символа подчеркивания (\_) не возвращаются пользователю. Это позволяет нам использовать имена файлов, чтобы различия представления, которые мы хотим показать, и файлы, которые их поддерживают. Имена макетов, которые являются поддерживающими файлами, начинаются с подчеркивания.

### Листинг 5-5: Начальное содержание макета

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

Макеты являются специализированной формой представления, и вы видите, что мы выделили в листинге выражения с @. Вызов метода `@RenderBody` вставляет содержимое представления, указанного методом действия в разметку макета. Другое Razor выражение в макете ищет свойство `Title` во `ViewBag` для того, чтобы установить содержание элемента `title`.

Любые элементы в макете будут применяться к любому представлению, которое использует макет, и именно поэтому макеты по существу являются шаблонами. В [листе 5-6](#) мы добавили немного простой разметки, чтобы показать, как это работает.

### Листинг 5-6: Добавление элементов в макет

```
<!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>
```

Мы добавили пару элементов заголовков и применили некоторые CSS стили к элементу `div`, который содержит выражение `@RenderBody`, просто чтобы было понятно, какой контент приходит из макета, а какой из представления.

## Применение макета

Чтобы применить макет к представлению, нам просто нужно установить значение свойства `Layout`. Мы также можем удалить элементы, которые обеспечивают структуру полной HTML страницы, потому что это будет выпадать из макета. Вы можете увидеть, как мы применили макет, в [листе 5-7](#), который показывает радикально упрощенный файл `Index.cshtml`.

### Совет

Мы также установили значение для свойства `ViewBag.Title`, которое будет использоваться в качестве контента элемента `title` в HTML документе, отправленном обратно пользователю: это не является обязательным, но это хорошая практика. Если значение для свойства не установлено, MVC фреймворк вернет пустой элемент `title`.

**Листинг 5-7:** Использование свойства Layout для определения файла представления

```
@model Razor.Models.Product
{
    ViewBag.Title = "Product Name";
    Layout = "~/Views/_BasicLayout.cshtml";
}
Product Name: @Model.Name
```

Преобразование довольно интересно, даже для такого простого представления. То, с чем мы остались, ориентировано на показе данных из объекта модели представления пользователю, а структура HTML документа исчезла.

Использование макетов имеет ряд преимуществ. Это позволяет упростить наши представления (что и показано в листинге), это позволяет нам создать общий HTML, который мы можем применить к нескольким представлениям, и, естественно, это сильно упрощает поддержку, потому что мы можем в одном месте изменить общий HTML и быть уверенными, что это изменение будет применяться везде, где используется макет. Чтобы увидеть результат использования макета, запустите приложение из примера. Результат представлен на [рисунке 5-7](#).

**Рисунок 5-7:** Результат применения простого макета к представлению



### Использование файла `_ViewStart`

Но есть еще кое-что, с чем надо разобраться: то есть, нам нужно указывать файл макета для каждого представления. Это обозначает, что если нам понадобится переименовать файл макета, нам нужно

будет найти каждое представление, которое к нему относится, и внести изменения. Это будет процесс, полный ошибок, который идет вразрез с общей темой простоты обслуживания, являющейся фактически лозунгом MVC фреймворка.

Эту проблему можно решить при помощи файла \_ViewStart.cshtml. При показе представления MVC фреймворк будет искать файл \_ViewStart.cshtml. Содержимое этого файла будет рассматриваться так, как если бы оно содержалось в самом файле представления, и мы можем использовать эту функцию, чтобы автоматически устанавливать значение свойства Layout.

Чтобы создать файл \_ViewStart.cshtml, добавьте новый файл макета в папку Views, повторив те действия, которые мы показали вам ранее. Назовите файл \_ViewStart.cshtml и измените его так, чтобы он соответствовал [листингу 5-8](#).

#### Листинг 5-8: Создание файла \_ViewStart.cshtml

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

Наш файл \_ViewStart.cshtml содержит значение для свойства Layout, что обозначает, что мы можем не использовать соответствующее выражение в файле Index.cshtml, как показано в [листинге 5-9](#).

#### Листинг 5-9: Обновление представления для демонстрации использования файла \_ViewStart.cshtml

```
@model Razor.Models.Product  
{  
    ViewBag.Title = "Product Name";  
}  
Product Name: @Model.Name
```

Нам совсем не нужно указывать, что мы хотим использовать файл \_ViewStart.cshtml. MVC фреймворк автоматически найдет файл и будет использовать его содержимое. Значения, определенные в файле представления, имеют преимущество, которое позволяет легко изменить файл \_ViewStart.cshtml.

#### *Внимание*

*Важно понимать разницу между тем, опускается ли свойство Layout в файле представления или устанавливается на null. Если ваше представление является автономным, и вы не хотите использовать макет, установите значение свойства Layout на null. Если же вы опустите свойство Layout, то MVC фреймворк посчитает, что вы хотите использовать макет и что он должен воспользоваться значением, которое найдет в файле \_ViewStart.cshtml.*

## Демонстрация макетов с общим доступом

Чтобы быстро и просто показать, как предоставить общий доступ для макетов, мы добавили новый метод действия NameAndPrice в контроллер Home. Вы можете посмотреть описание этого метода в листинге 5-10, который демонстрирует изменения, внесенные в файл /Controllers/HomeController.cs.

#### Листинг 5-10: Добавление нового метода действия в контроллер Home

```

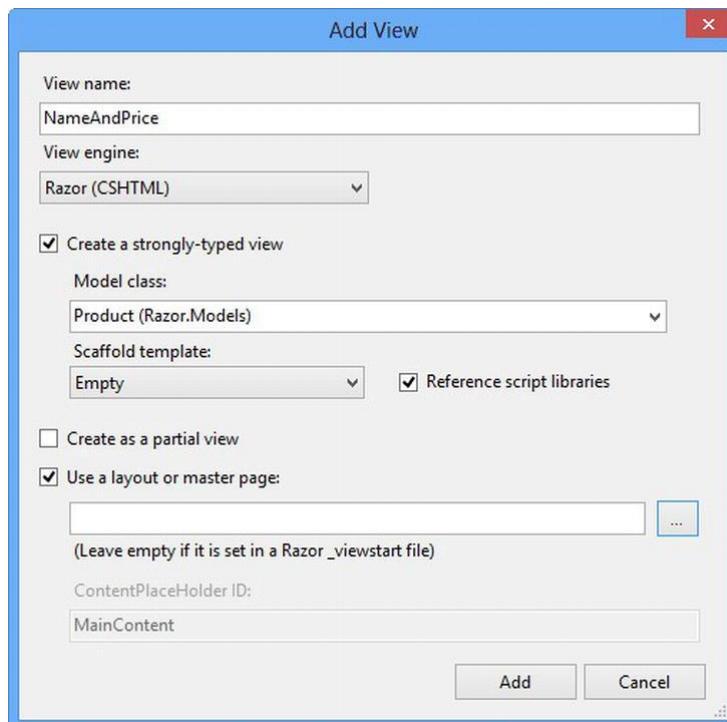
using Razor.Models;
using System;
using System.Web.Mvc;
namespace Razor.Controllers
{
    public class HomeController : Controller
    {
        Product myProduct = new Product
        {
            ProductID = 1,
            Name = "Kayak",
            Description = "A boat for one person",
            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, так же как это делает метод действия Index: это не то, что вы могли бы сделать в реальном проекте, но мы демонстрируем функциональность Razor, и такой очень простой пример подходит для наших потребностей.

Щелкните правой кнопкой мыши в редакторе по методу NameAndPrice и выберите из всплывающего меню пункт Add View для отображения диалогового окна Add View. Поставьте галочку на Create a strongly-typed view и выберите из раскрывающегося списка класс Product. Поставьте галочку на Use a layout or master page, как показано на [рисунке 5-8](#).

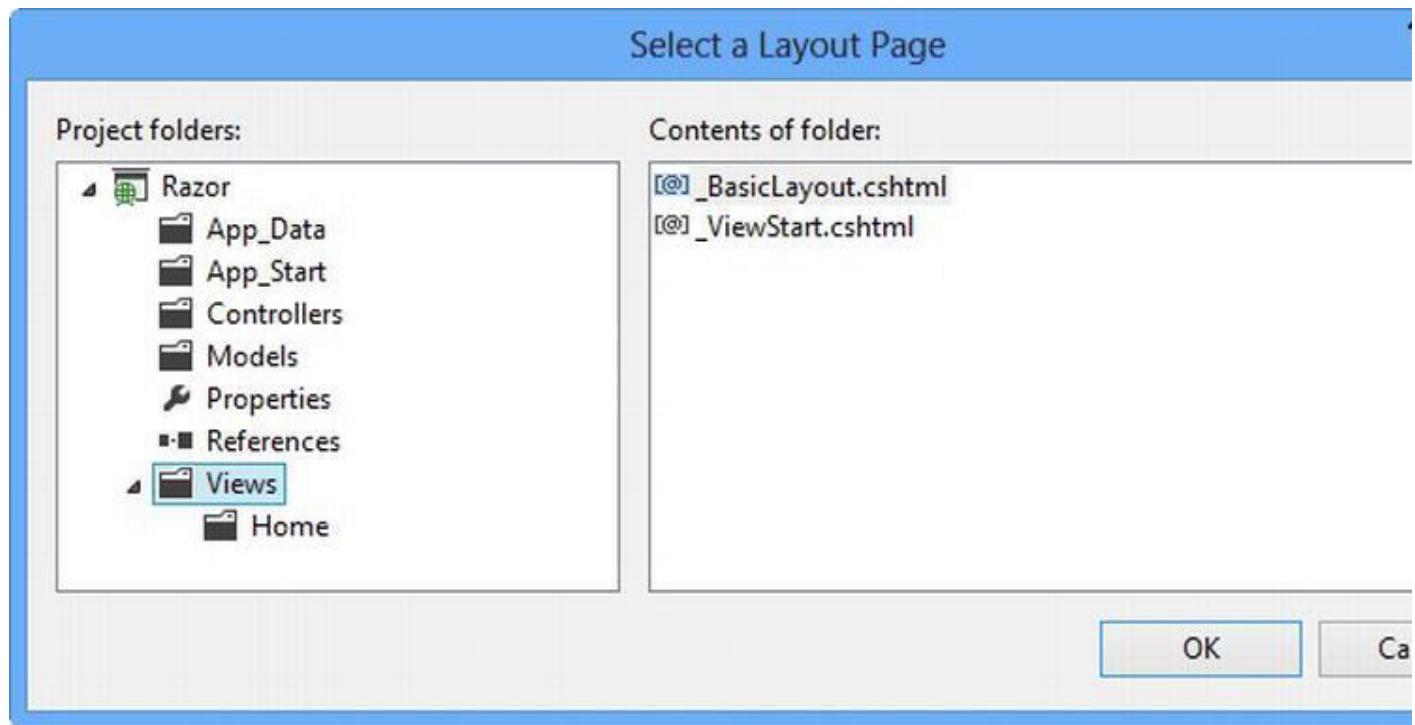
**Рисунок 5-8:** Создание представления, которое использует макет



Обратите внимание на текст под опцией `Use a layout`. Он говорит, что вы должны оставить текстовое поле пустым, если вы указали представление, которое вы хотите использовать в файле `_ViewStart.cshtml`. Если вы нажмете на этом пункте кнопку `Add`, будет создано представление без оператора C#, который устанавливает значение для свойства `Layout`.

Мы собираемся явно указать представление, поэтому щелкните по кнопке с многоточием (...), которая находится справа от текстового поля. Visual Studio покажет вам диалоговое окно, которое позволяет выбрать файл макета, как показано на [рисунке 5-9](#).

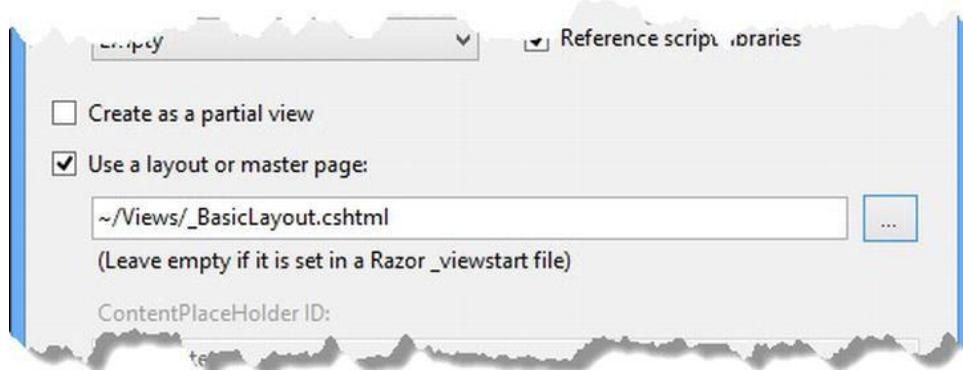
**Рисунок 5-9:** Выбор файла макета



Соглашение для MVC проектов заключается в размещение файла макета в папке `Views`, и поэтому диалоговое окно представляет для выбора содержимое именно этой папки. Но это всего лишь соглашение, и поэтому левая панель диалогового окна позволяет перемещаться по проекту, если вдруг вы решили не следовать соглашению.

Мы определили только один файл макета, поэтому выберите `_BasicLayout.cshtml` и нажмите кнопку **OK**, чтобы вернуться в диалоговое окно `Add View`. Вы увидите, что имя файла макета было помещено в текстовое поле, как показано на [рисунке 5-10](#).

**Рисунок 5-10:** Определение файла макета при создании представления



Нажмите кнопку Add для создания файла /Views/Home/NameAndPrice.cshtml. Содержимое этого файла представлено в [листинге 5-11](#).

#### Листинг 5-11: Содержимое представления NameAndPrice

```
@model Razor.Models.Product
{
    ViewBag.Title = "NameAndPrice";
    Layout = "~/Views/_BasicLayout.cshtml";
}
<h2>NameAndPrice</h2>
```

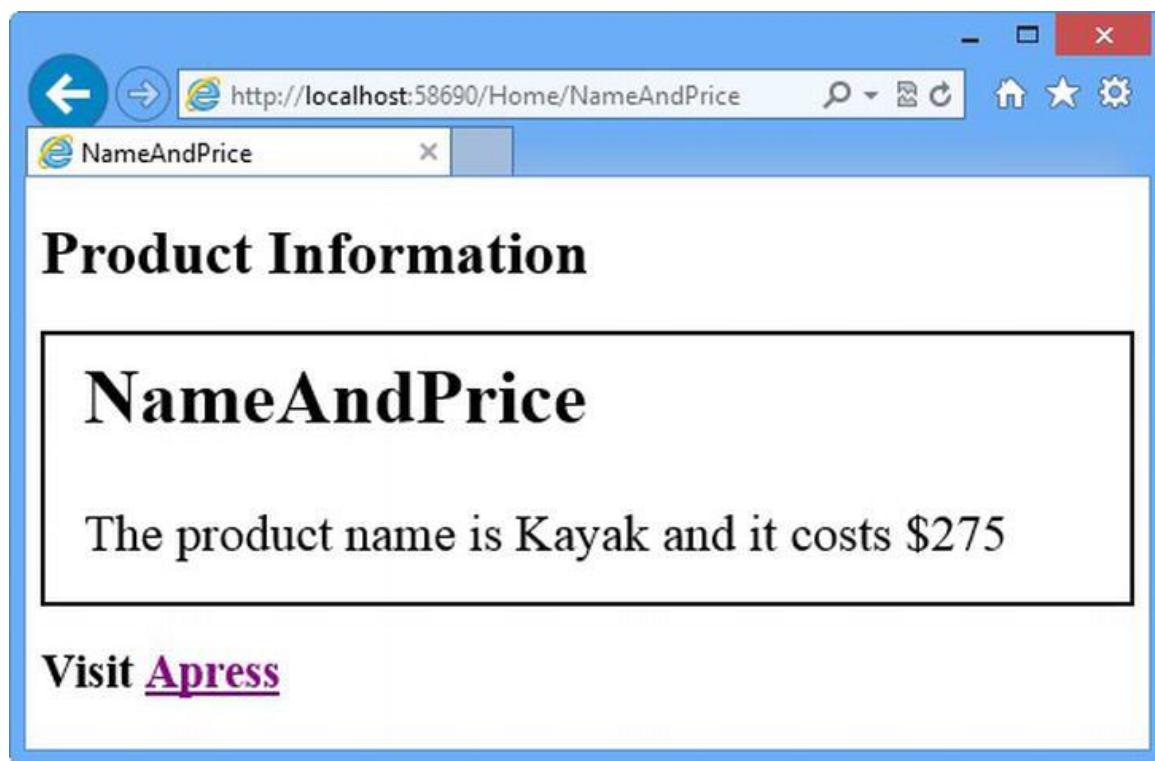
Visual Studio использует немного другой контент по умолчанию для файлов представления, если вы указываете макет, но вы видите, что результат содержит те же Razor выражения, которые мы использовали, когда сами применяли макет к представлению. Чтобы завершить этот пример, мы добавили в [листинг 5-12](#) простое дополнение к файлу NameAndPrice.cshtml, где отображено значение данных объекта модели представления.

#### Листинг 5-12: Дополнения в макет 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 к /Home/NameAndPrice, вы увидите результаты, показанные на рисунке 5-11. Как вы и ожидали, общие элементы и стили, определенные в макете, были применены к представлению. Это показывает, как макет может быть использован в качестве шаблона для создания общего внешнего вида приложения (хотя он довольно простой и не совсем привлекательный).

Рисунок 5-11: Содержание файла макета, применяемое к представлению NameAndPrice



### Примечание

У нас был бы тот же результат, если бы мы оставили текстовое поле в диалоговом окне Add View пустым и работали с файлом \_ViewStart.cshtml. Мы указали файл явно только потому, что мы хотели показать вам функцию Visual Studio, которая помогает выбрать необходимый файл.

## Использование выражений Razor

Теперь, когда мы показали вам основы представлений и макетов, мы собираемся обратиться к различным видам выражений, которые поддерживает Razor, и рассказать вам, как их использовать для создания контента представления.

В хорошем MVC приложении существует четкое разделение между ролями, которые выполняют метод действия и представление. Для этой главы правила просты, и мы свели их в [таблицу 5-1](#).

**Таблица 5-1:** Роли, выполняемые методом действия и представлением

Компонент	Выполняет	Не выполняет
Action	Передает объект модели представления в представление	Не передает отформатированные данные в представление
Method		
View	Использует объект модели представления для показа контента пользователю	Не меняет ни один из аспектов объекта модели представления

Мы будем возвращаться к этой теме снова и снова на протяжении всей книги. Чтобы получить все лучшее от MVC фреймворка, вам нужно уважать разделение различных частей приложения и соблюдать его. Вы увидите, что многое можно сделать с Razor, в том числе с использованием операторов C#, но вы ни в коем случае не должны использовать Razor для выполнения операций с бизнес логикой или для манипулирования объектами доменной модели.

Равным образом, вы не должны форматировать данные, которые ваш метод действия передает представлению. Вместо этого, позвольте представлению вычислить данные, которые оно должно отобразить. Вы видели очень простой пример этого в предыдущих разделах данной главы. Мы определили метод действия `NameAndPrice`, который отображает значения свойств `Name` и `Price` объекта `Product`. И хотя мы знали, какие именно свойства нам нужно отобразить, мы передали весь объект `Product` модели представления, вот так:

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

Затем в представлении мы использовали Razor выражение `@Model`, чтобы получить значения интересующих нас свойств:

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

Мы могли бы создать строку, которую хотим отобразить, в методе действия и передать ее в качестве объекта модели представления в представление. Это сработало бы, но такой подход подрывает преимущество MVC паттерна и уменьшает возможность реагировать на изменения в будущем. Как мы уже говорили, мы вернемся к этой теме снова, но вы должны знать, что MVC фреймворк не навязывает надлежащее использование MVC паттерна, и вы должны постоянно помнить, к какому результату могут привести созданный вами код и дизайнерские решения.

## Вставка значений данных

Самое простое, что вы можете сделать с Razor выражением, это вставить значение данных в разметку. Вы можете сделать это, используя выражение `@Model`, чтобы обратиться к свойствам и методам, определенным в модели представления. Или же вы можете использовать выражение `@ViewBag`, чтобы обратиться к свойствам, которые вы определили динамически при помощи контейнера `ViewBag` (мы представили ее в [главе 2](#)).

Вы уже видели примеры обоих этих выражений, но для полноты картины мы добавили новый метод действия `DemoExpressions` в контроллер `Home`, который передает данные в представление, используя объект модели и `ViewBag`. Вы можете ознакомиться с определением нового метода действия в [листинге 5-13](#).

### Листинг 15-13: 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);
        }
    }
}
```

Мы создали строго типизированное представление `DemoExpression.cshtml`, чтобы показать эти основные типы выражения данных. Вы можете увидеть содержимое файла представления в [листинге 5-14](#).

### Листинг 5-14: Содержание файла представления `DemoExpression`

```
@model Razor.Models.Product
 @{
    ViewBag.Title = "DemoExpression";
}
<table>
    <thead>
```

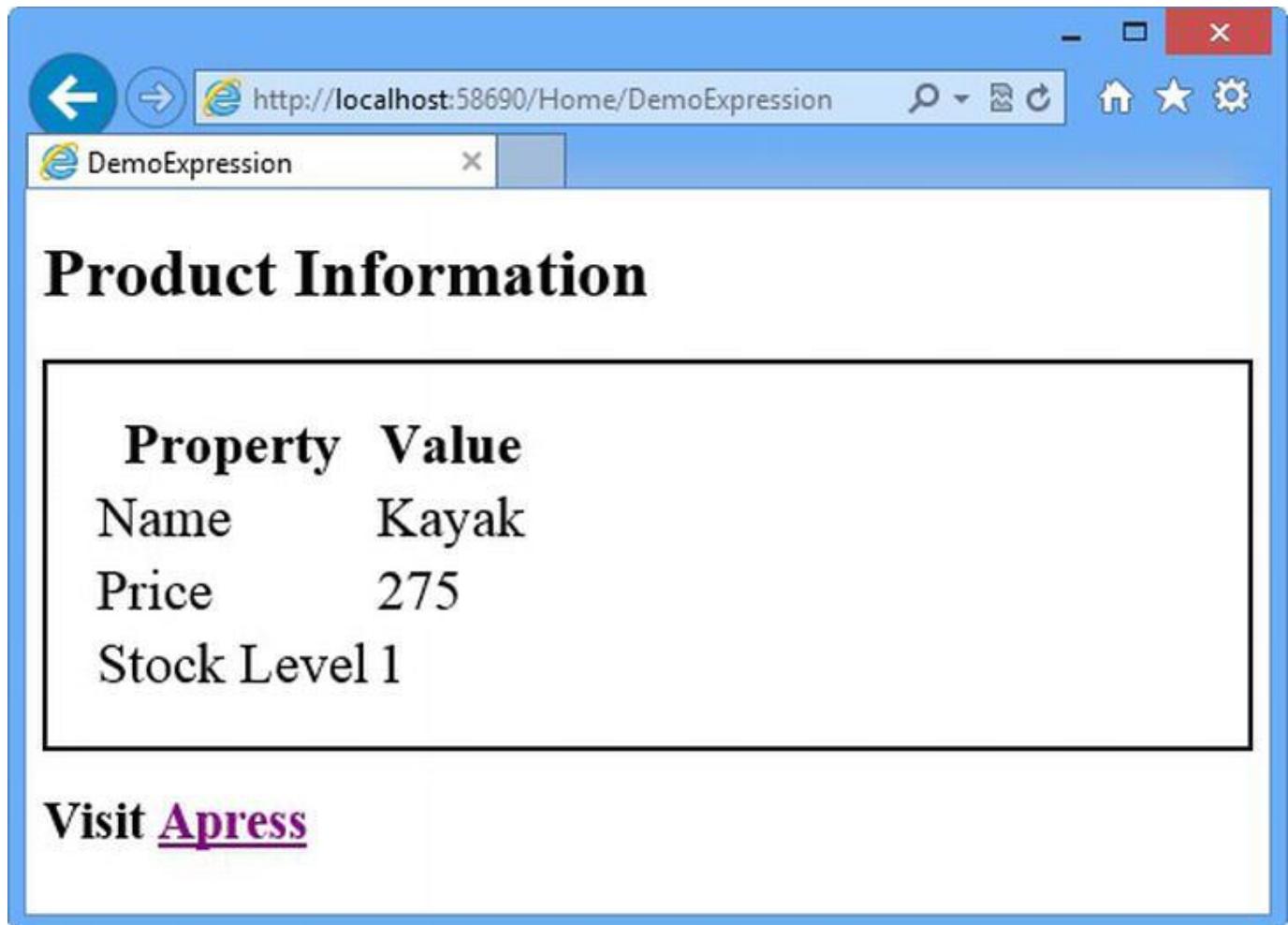
```

<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>

```

Для этого примера мы создали простую HTML таблицу и использовали свойства объекта модели и ViewBag для заполнения ячеек значениями. На [рисунке 5-12](#) вы можете увидеть результат запуска приложений и переход по адресу /Home/DemoExpression. Это всего лишь еще одна демонстрация основных Razor выражений, которые мы до сих пор использовали в примерах.

**Рисунок 5-12:** Использование основных выражений Razor для вставки значений данных в HTML разметку



Результат не особо красивый, потому что мы не применяли никаких CSS стилей к HTML элементам, которые генерируют представление и макет. Однако этот пример служит для четкого понимания того, каким образом могут быть использованы основные Razor выражения для отображения данных, передаваемых от метода действия к представлению.

## Установка значений атрибутов

Все наши примеры до сих пор касались содержания элементов, но вы также можете использовать Razor выражения для установки значений атрибутов элементов. В [листе 5-15](#) показано, как мы изменили представление DemoExpression, чтобы использовать ViewBag свойства для значений атрибутов. Способ, которым Razor обрабатывает атрибуты в MVC 4, довольно разумный и интересный, и это одна из тех областей, которые были улучшены после MVC 3.

### Листинг 5-15: Использование Razor выражения для установки значения атрибута

```
@model Razor.Models.Product
{
    ViewBag.Title = "DemoExpression";
    Layout = "~/Views/_BasicLayout.cshtml";
}


| Property    | Value                 |
|-------------|-----------------------|
| Name        | @Model.Name           |
| Price       | @Model.Price          |
| Stock Level | @ViewBag.ProductCount |



The containing element has data attributes


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

Мы начали с использования основных Razor выражений для установки значений некоторых атрибутов data элемента div. data-атрибуты, которые являются атрибутами, чьи имена начинаются с data-, были неформальным способом создания пользовательских атрибутов в течение многих лет и стали стандартом в HTML5. Мы использовали значения свойств ViewBag ApplyDiscount, ExpressShip и Supplier, чтобы установить значения этих атрибутов.

Запустите пример приложения, выберите нужный метод действия и посмотрите на исходный HTML, который был использован для отображения страницы. Вы увидите, что Razor установил значения атрибутов следующим образом:

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

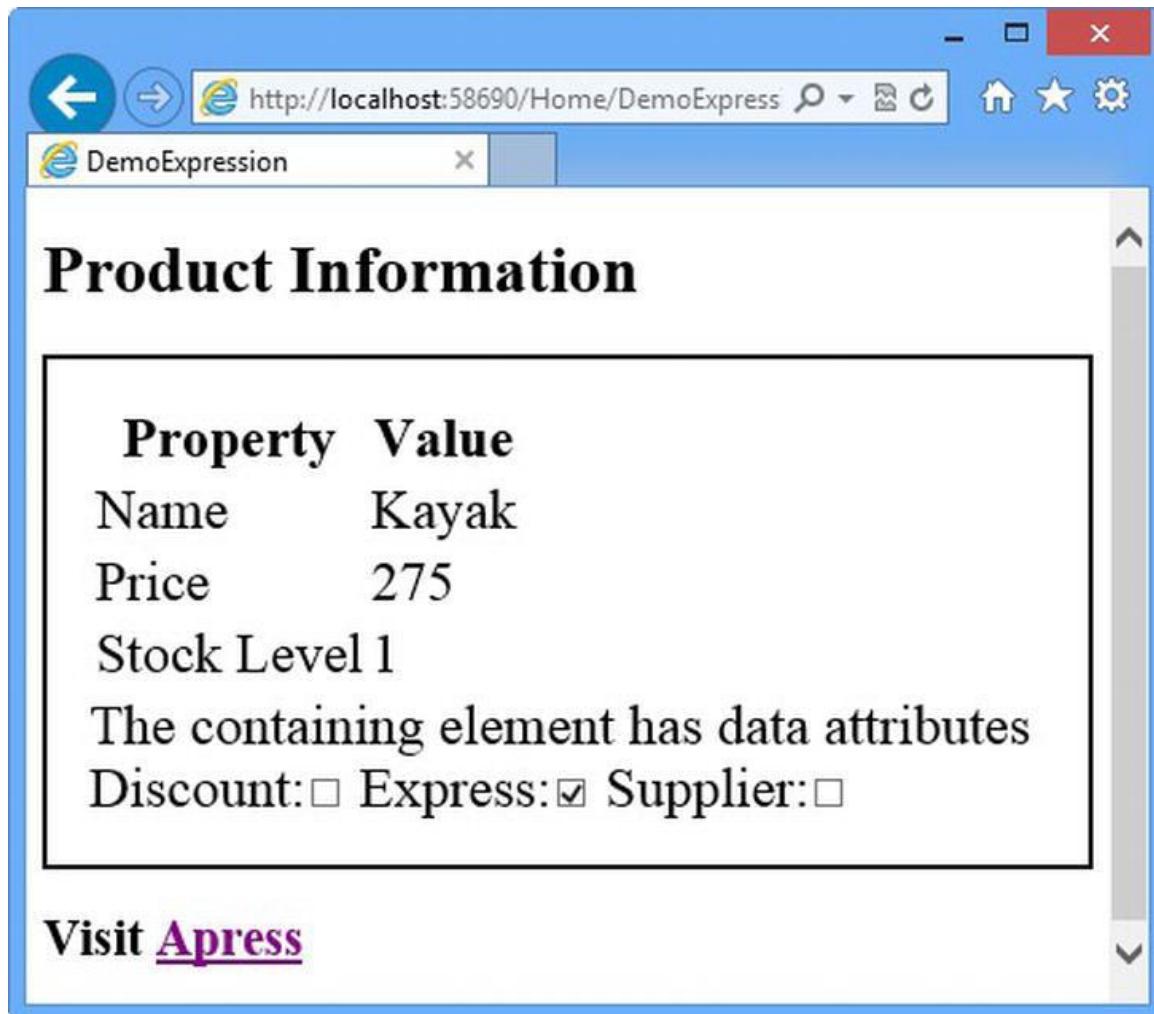
Значения `False` и `True` соответствуют булевым `viewBag` значениям, но Razor сделал кое-что разумное для свойства, чье значение равно `null`, а именно, представил пустую строку.

Но все становится гораздо более интересным, когда мы посмотрим на второе наше дополнение в представлении: это набор чекбоксов, чей атрибут `checked` установлен на те же самые `ViewBag` свойства, которые мы использовали для атрибутов `data`. HTML, который представил Razor, выглядит следующим образом:

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

В MVC 4 Razor стал понимать, как используется такие атрибуты как `checked`, где наличие атрибута, а не его значение, меняет конфигурацию элемента. Если бы Razor вставили `False`, или `null`, или пустую строку в качестве значения атрибута `checked`, тогда в чекбоксе, который отображает браузер, стояла бы галочка. Вместо этого Razor полностью удаляет атрибут из элемента, если значение равно `false` или `null`, создавая нужный результат, как показано на [рисунке 5-13](#).

Рисунок 5-13: Результат удаления атрибутов, чье присутствие влияет на элемент



## Использование условных операторов

Razor способен обрабатывать условные операторы, что обозначает, что мы можем подгонять выходные данные наших представлений, основываясь на значениях данных в нашем представлении. Мы собираемся добраться до самого сердца Razor, что позволяет нам создавать сложные и гибкие макетов, которые остаются простыми для понимания и поддержки. В листинге 5-16 мы обновили наш файл представления `DemoExpression.cshtml`, чтобы включить в него условный оператор.

### Листинг 5-16: Использование условного оператора Razor

```
@model Razor.Models.Product
{
    ViewBag.Title = "DemoExpression";
    Layout = "~/Views/_BasicLayout.cshtml";
}


| Property    | Value                                                                                                                                                                                                                                                                    |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Name        | @Model.Name                                                                                                                                                                                                                                                              |
| Price       | @Model.Price                                                                                                                                                                                                                                                             |
| Stock Level | @switch ((int)ViewBag.ProductCount) {         case 0:             @: Out of Stock             break;         case 1:             <b>Low Stock (@ViewBag.ProductCount)</b>             break;         default:             @ViewBag.ProductCount             break;     } |


```

Чтобы написать условный оператор, поместите символ `@` в начале этого оператора, в данном примере перед ключевым словом C# `switch`. Блок кода заканчивается фигурной скобкой `()` так же, как и обычный блок кода C#.

#### Совет

*Обратите внимание, что мы привели значение свойства `ViewBag.ProductCount` к `int`, чтобы использовать его с оператором `switch`. Это необходимо, поскольку оператор `switch` работает только с определенным набором типов, и он не может оценить динамическое свойство без приведения типов.*

Внутри блока кода Razor вы можете вставлять HTML элементы и значения данных для представления, просто определив HTML и Razor выражения вот таким образом:

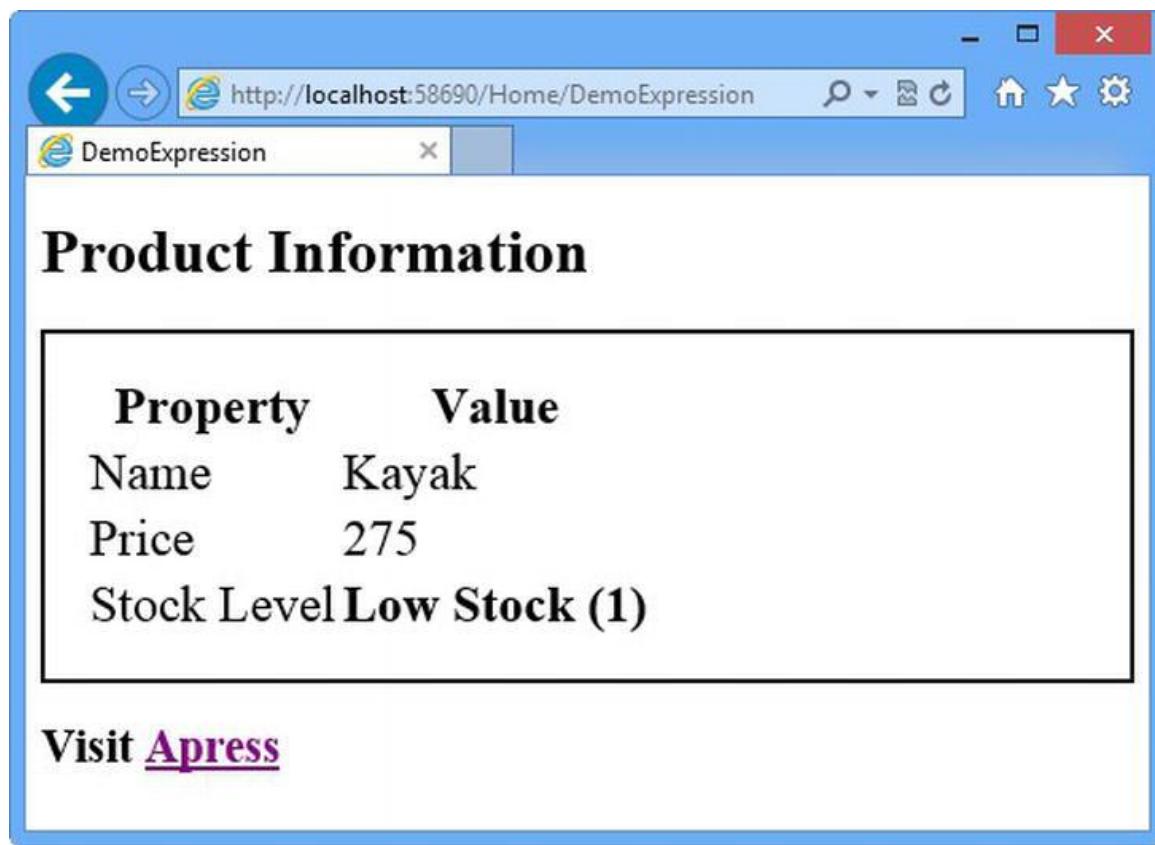
```
...
<b>Low Stock (@ViewBag.ProductCount)</b>
...
@ViewBag.ProductCount
...
```

Нам не нужно заключать элементы или выражения в кавычки или как-то по-особенному обозначать их: движок Razor будет интерпретировать их как выходные данные, которые будут обрабатываться в обычном порядке. Однако если вы хотите вставить в представление просто текст, когда он не содержится в HTML элементе, вы должны помочь Razor и обозначить строку следующим образом:

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

Символ `@:` предотвращения Razor от интерпретации строки как C# выражения, что является поведением по умолчанию, когда он сталкивается с текстом. Вы можете увидеть результат использования нашего условного оператора на [рисунке 5-14](#).

**Рисунок 5-14:** Использование условного оператора `switch` в представлении Razor



Условные операторы играют важную роль в Razor представлениях, потому что они позволяют подгонять содержание к значениям данных, которые представление получает от метода действия. В качестве дополнительного примера в [листинге 5-17](#) показано добавление оператора `if` к представлению `DemoExpression.cshtml` – вы, естественно, знаете, что это очень широко используемый условный оператор.

### Листинг 5-17: Использование условного 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 дает тот же результат, что и `switch`, но мы просто хотели показать, как вы можете связывать условные операторы C# с Razor представлениями. Мы объясняем, как это все работает, в главе 18, когда мы более углубленно рассмотрим представления.

### Перечисление содержимого массивов и коллекций

При написании MVC приложений вам часто будет нужно перечислять содержимое массива или другого вида коллекции объектов, и генерировать подробный контент каждого из них. Чтобы показать, как это делается, мы определили новый метод действия `DemoArray` в контроллере `Home`, который вы можете увидеть в [листинге 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",
        }
    }
}
```

```

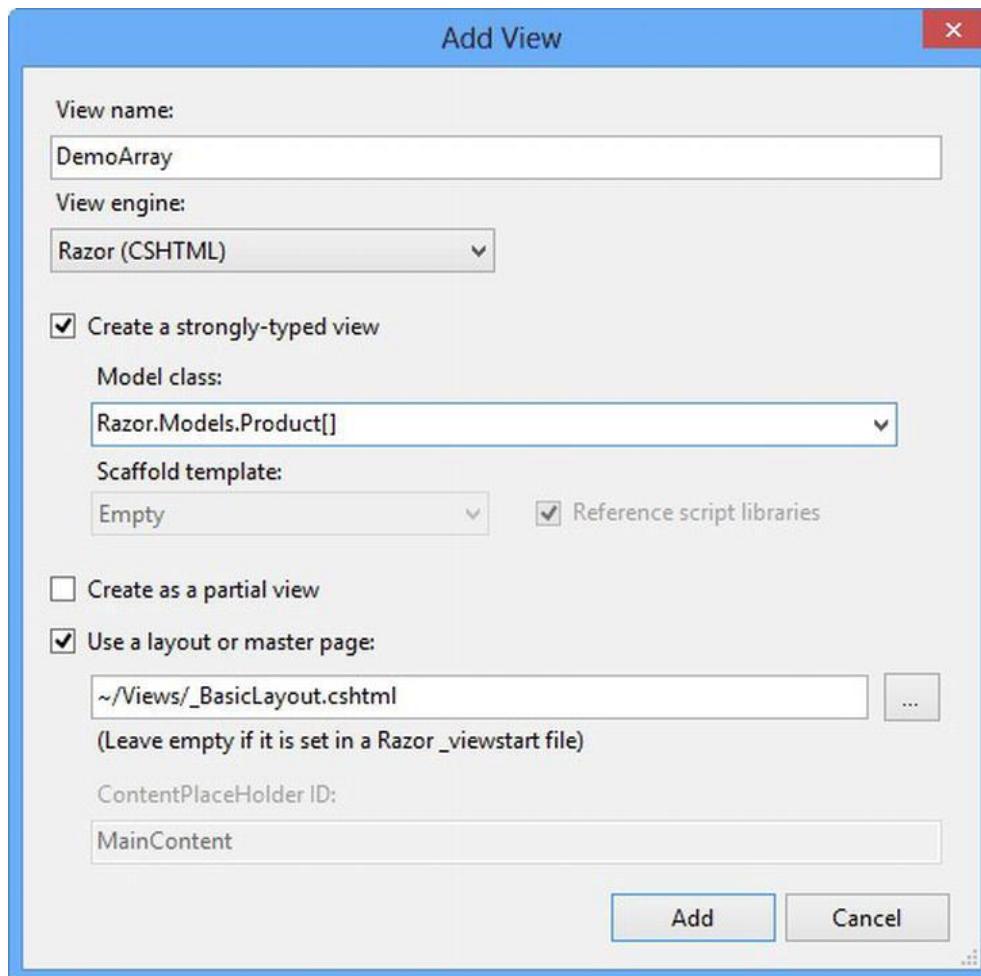
        Category = "Watersports",
        Price = 275M
    };
    // ...другие методы действия опущены для краткости...
public ActionResult DemoArray()
{
    Product[] array = {
        new Product {Name = "Kayak", Price = 275M},
        new Product {Name = "Lifejacket", Price = 48.95M},
        new Product {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 несколько простых значений данных, и передает его методу `View`, и эти данные показываются при помощи представления по умолчанию.

Visual Studio не предлагает опции для массивов и коллекций при создании представления, поэтому вам придется вручную ввести информацию о требуемом типе в диалоговом окне `Add View`. Вы можете увидеть, как мы сделали это, на [рисунке 5-15](#). Тут показано, как мы создали представление `DemoArray.cshtml` и указали, что тип модели представления – это `Razor.Models.Product []`.

**Рисунок 5-15:** Установка вручную типа модели представления для строго типизированного представления



Вы можете увидеть содержимое файла представления `DemoArray.cshtml` в [листе 5-19](#), куда включены дополнения, которые мы сделали, чтобы показать информацию по элементам массива пользователям.

**Листинг 5-19:** Содержимое файла `DemoArray.cshtml`

```
@model Razor.Models.Product[]
{
    ViewBag.Title = "DemoArray";
    Layout = "~/Views/_BasicLayout.cshtml";
}
@if (Model.Length > 0) {

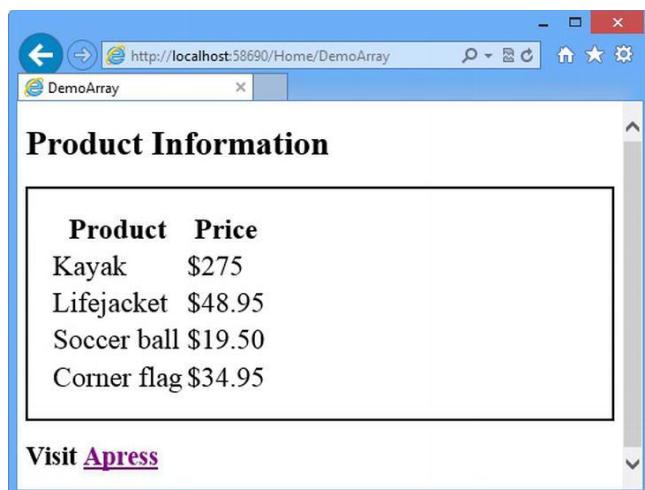

| Product | Price      |
|---------|------------|
| @p.Name | \$@p.Price |


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

Мы использовали оператор `@if`, чтобы менять содержание в зависимости от длины массива, с которым мы работаем, и оператор `@foreach`, чтобы перечислить содержимое массива и создать строку в HTML таблицу для каждого из элементов. Вы видите, как эти выражения соответствуют своим C# копиям, и как мы создали локальную переменную `p` в цикле `foreach`, а затем обратились к ее свойствам, используя Razor выражения `@p.Name` и `@p.Price`.

В результате мы получаем элемент `h2`, если массив пуст, или, в противном случае, создаем одну строку для каждого элемента массива в HTML таблице. Поскольку в этом примере наши данные являются статическими, вы всегда будете видеть один и тот же результат, который представлен на рисунке 5-16.

**Рисунок 5-16:** Генерирование элементов при помощи оператора `foreach`



## Работа с пространством имен

Вы заметили, что в последнем примере мы должны были обращаться к классу `Product` в цикле `foreach` по его полному имени, вот таки образом:

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

Это может раздражать в сложных представлениях, где у вас будет много ссылок на модели представления и другие классы. Мы можем привести в порядок представления при помощи выражения `@using`, чтобы вставить пространство имен в контекст, как и для обычного C# класса. В [листе 5-20](#) показано, как мы применили выражение `@using` для представления `DemoArray.cshtml`, которое мы создали ранее.

**Листинг 5-20:** Применение выражения `@using`

```
@using Razor.Models
@model Product[]
{
    ViewBag.Title = "DemoArray";
    Layout = "~/Views/_BasicLayout.cshtml";
}
@if (Model.Length > 0) {


| Product | Price      |
|---------|------------|
| @p.Name | \$@p.Price |


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

Представление может содержать несколько выражений `@using`. Мы использовали выражение `@using`, чтобы импортировать пространство имен `Razor.Models`, что обозначает, что мы можем удалить пространство имен из выражения `@model` и из цикла `foreach`.

## Резюме

В этой главе мы показали вам движок представления Razor и как он может быть использован для генерации HTML. Мы показали вам, как обратиться к данным, передаваемым из контроллера через объект модели представления и `ViewBag`, и мы показали вам, как Razor выражения могут быть использованы для создания ответов для пользователя на основе данных, с которыми вы работаете. Вы увидите много различных примеров того, как может быть использован Razor, в остальной части книги, и мы подробно расскажем, как работает механизм представлений MVC, в главе 18. В следующей главе мы опишем важные инструменты разработки и тестирования, которые лежат в основе MVC фреймворка и которые помогут вам существенно улучшить ваши проекты.\

# Важные инструменты MVC

В этой главе мы рассмотрим три инструмента, которые должны быть в арсенале каждого MVC программиста: контейнер внедрения зависимостей (DI), фреймворк для модульного тестирования и инструмент для мокинга (mock-объектов).

Мы выбрали три конкретные реализации этих инструментов для данной книги, но есть много альтернатив для каждого типа инструмента. Если вам не нравятся те, которые мы используем, не волнуйтесь. Их так много, что вы наверняка найдете то, что подходит именно вам и облегчит ваш рабочий процесс.

Как мы уже отмечали в [главе 3](#), Ninject – наш предпочтаемый DI контейнер. Он простой, элегантный и легкий в использовании. Есть более сложные альтернативы, но нам нравится способ, которым работает Ninject, причем, с минимумом настроек. Мы считаем отправной точкой паттерны, а не закон, и мы нашли что легко подгонять наш DI с Ninject к различным проектам и рабочим процессам. Если вам не нравится Ninject, мы рекомендуем попробовать Unity, который является одной из альтернатив Microsoft.

Для модульного тестирования мы будем использовать то, что встроено в Visual Studio. Мы привыкли использовать NUnit, который является самым популярным .NET фреймворком для модульного тестирования, но Microsoft сделал большой рывок в улучшении поддержки модульного тестирования в Visual Studio (и в настоящее время включает ее в бесплатное выпуск Visual Studio). В результате этого фреймворки для модульного тестирования, тесно интегрированы в остальную часть IDE и они стали довольно хорошими.

Третий инструмент, выбранный нами, это Moq, который является набором мокинг инструментов. Мы используем Moq в создании реализаций интерфейсов для использования в наших модульных тестах. Программисты или любят, или ненавидят Moq; третьего не дано. Либо вы посчитаете синтаксис элегантным и выразительным, или вы будете проклинать его каждый раз при попытке использования. Если вам он не по душе, обратите внимание на Rhino Mocks, который является хорошей альтернативой.

Мы расскажем о каждом из этих инструментов и покажем их основные возможности. Мы не дадим исчерпывающей информации по этим инструментам, ведь по каждому из них можно написать отдельную книгу, но мы дадим вам достаточно информации, чтобы вы могли начать работу с ними, и, что особенно важно, следовать примерам в остальной части книги.

## Примечание

*В этой главе предполагается, что вы захотите получить все преимущества от MVC, включая архитектуру, которая поддерживает множество тестов, и тут мы сделаем акцент на создании приложений, которые легко изменяются и поддерживаются. Мы любим эти инструменты и не будем без них создавать приложения, но мы знаем, что некоторые читатели просто хотят понять возможности, которые предлагает MVC, и не углубляться в развитие философии и методологии. Мы не будем пытаться вас разубедить – это ваше личное решение, и вы знаете, что вам нужно, чтобы наиболее эффективно создавать свои проекты. Мы предполагаем, что вы, по крайней мере, бегло пробежите по этой главе, чтобы увидеть доступные возможности, но если вы не относитесь к типу любителей юнит тестирования, то можете сразу перейти к следующей главе и посмотреть, как построить реальное MVC приложение.*

# Создание проекта для примера

Мы собираемся начать с создания простого проекта для примера, который мы будем использовать в этой главе. Мы создали новый Visual Studio проект, используя шаблон ASP.NET MVC 4 Web Application и выбрав опцию Empty для создания MVC проекта без начального содержания. Это такой же проект, с каким мы до сих пор работали, и мы назвали проект для этого примера EssentialTools.

## Создание классов модели

Затем добавьте файл для класса `Product.cs` в папку проекта `Models` и установите контент в соответствии с [листигом 6-1](#). Это тот же самый класс модели, который использовался в предыдущих главах, и единственное изменение заключается в том, чтобы пространство имен соответствовало проекту `EssentialTools`.

### Листинг 6-1: Класс `Product`

```
namespace EssentialTools.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; }
    }
}
```

Мы также хотим добавить класс, который будет рассчитывать общую стоимость объектов `Product` (коллекция объектов). Добавьте новый файл класса `LinqValueCalculator.cs` в папку `Models` и задайте ему содержание в соответствии с [листигом 6-2](#).

### Листинг 6-2: Класс `LinqValueCalculator`

```
using System.Collections.Generic;
using System.Linq;
namespace EssentialTools.Models
{
    public class LinqValueCalculator
    {
        public decimal ValueProducts(IEnumerable<Product> products)
        {
            return products.Sum(p => p.Price);
        }
    }
}
```

Класс `LinqValueCalculator` определяет единственный метод `ValueProducts`, использующий LINQ метод `Sum`, чтобы сложить значения свойства `Price` каждого объекта `Product`, который передается методу (приятная функция LINQ, которой мы часто пользуемся).

Наш последний класс модели называется `ShoppingCart`, он представляет собой коллекцию объектов `Product` и использует `LinqValueCalculator` для определения их общей стоимости. Создайте новый файл класса `ShoppingCart.cs` и убедитесь, что его содержимое соответствуют [листигу 6-3](#).

### Листинг 6-3: Класс ShoppingCart

```
using System.Collections.Generic;
namespace EssentialTools.Models
{
    public class ShoppingCart
    {
        private LinqValueCalculator calc;
        public ShoppingCart(LinqValueCalculator calcParam)
        {
            calc = calcParam;
        }
        public IEnumerable<Product> Products { get; set; }
        public decimal CalculateProductTotal()
        {
            return calc.ValueProducts(Products);
        }
    }
}
```

### Добавление контроллера

Добавьте новый контроллер `HomeController` в папку `Controllers` и установите его контент в соответствии с [листингом 6-4](#). Метод действия `Index` создает массив объектов `Product` и использует класс `LinqValueCalculator` для получения общей стоимости, которая передается методу `View`. Мы не указываем представление, когда вызываем метод `View`, поэтому будет использоваться представление по умолчанию.

### Листинг 6-4: Контроллер HomeController

```
using EssentialTools.Models;
using System.Web.Mvc;
using System.Linq;
namespace EssentialTools.Controllers
{
    public class HomeController : Controller
    {
        private Product[] products =
        {
            new Product {Name = "Kayak", Category = "Watersports", Price = 275M},
            new Product {Name = "Lifejacket", Category = "Watersports", Price = 48.95M},
            new Product {Name = "Soccer ball", Category = "Soccer", Price = 19.50M},
            new Product {Name = "Corner flag", Category = "Soccer", Price = 34.95M}
        };
        public ActionResult Index()
        {
            LinqValueCalculator calc = new LinqValueCalculator();
            ShoppingCart cart = new ShoppingCart(calc) { Products = products };
            decimal totalValue = cart.CalculateProductTotal();
            return View(totalValue);
        }
    }
}
```

### Добавление представления

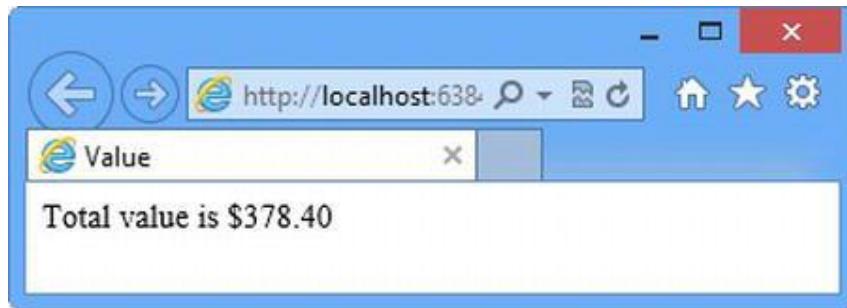
Последним дополнением к проекту является представление. Щелкните правой кнопкой мыши по методу действия `Index` в контроллере `Home`, а затем выберите `Add View`. Создайте представление `Index`, которое является строго типизированным с объектом модели представления `decimal`. Как только вы создадите файл, поменяйте его содержимое, чтобы оно соответствовало [листингу 6-5](#).

### Листинг 6-5: Файл Index.cshtml

```
@model decimal
{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Value</title>
</head>
<body>
    <div>
        Total value is $@Model
    </div>
</body>
</html>
```

Это очень простое представление, которое использует выражение @Model для отображения значения decimal, которое передается методом действия. Если вы запустите проект, вы увидите общую стоимость, подсчитанную классом LinqValueCalculator, что показано на [рисунке 6-1](#).

**Рисунок 6-1:** Запуск приложения из примера



Это очень простой проект, но это закладывает базу для изучения различных инструментов и методов, которые мы описываем в этой главе.

## Использование Ninject

Мы ввели понятие DI в главе 3. Напомним, что идея состоит в том, чтобы разделять компоненты в наших MVC приложениях, и мы делаем это, сочетая интерфейсы и DI. В следующих разделах мы разъясним проблему, которую мы сознательно создали в приложении из примера, и покажем, как использовать Ninject, наш любимый DI пакет, который может быть использован для ее решения.

### В чем же проблема?

В приложении мы создали одну проблему, которую может решить DI. Наш простой проект опирается на сильно связанные классы: класс ShoppingCart тесно связан с классом LinqValueCalculator, а класс HomeController тесно связан с обоими классами ShoppingCart и LinqValueCalculator. Это обозначает, что если мы захотим заменить класс LinqValueCalculator, нам нужно будет найти и заменить все ссылки в классах, которые тесно с ним связаны. В принципе, это не проблема для такого простого проекта, но это может принести много ошибок в реальный проект, особенно если мы хотим реализовать разные виды расчетов, а не просто заменить один класс другим.

## Применение интерфейса

Мы можем частично решить эту проблему, используя C# интерфейс для того, чтобы отделить определение функционала калькулятора от его реализации. Чтобы показать это, мы добавили файл `IValueCalculator.cs` в папку `Models` и создали интерфейс, показанный в [листе 6-6](#).

### Листинг 6-6: Интерфейс `IValueCalculator`

```
using System.Collections.Generic;
namespace EssentialTools.Models
{
    public interface IValueCalculator
    {
        decimal ValueProducts(IEnumerable<Product> products);
    }
}
```

Затем мы можем реализовать этот интерфейс в классе `LinqValueCalculator`, как показано в [листе 6-7](#).

### Листинг 6-7: Применение интерфейса к классу `LinqValueCalculator`

```
using System.Collections.Generic;
using System.Linq;
namespace EssentialTools.Models
{
    public class LinqValueCalculator : IValueCalculator
    {
        public decimal ValueProducts(IEnumerable<Product> products)
        {
            return products.Sum(p => p.Price);
        }
    }
}
```

Интерфейс позволяет нам сломать сильную связь между классами `ShoppingCart` и `LinqValueCalculator`, как показано в [листе 6-8](#).

### Листинг 6-8: Применение интерфейса к классу `ShoppingCart`

```
using System.Collections.Generic;
namespace EssentialTools.Models
{
    public class ShoppingCart
    {
        private IValueCalculator calc;
        public ShoppingCart(IValueCalculator calcParam)
        {
            calc = calcParam;
        }
        public IEnumerable<Product> Products { get; set; }
        public decimal CalculateProductTotal()
        {
            return calc.ValueProducts(Products);
        }
    }
}
```

Мы добились определенного прогресса, но C# требует от нас указать класс реализации для интерфейса при создании экземпляра, что достаточно справедливо, ведь ему необходимо знать, какой именно класс реализации мы хотим использовать. Это обозначает, что у нас до сих пор есть проблема в контроллере `Home`, когда мы создаем объект `LinqValueCalculator`:

```

...
public ActionResult Index() {
    IValueCalculator calc = new LinqValueCalculator();
    ShoppingCart cart = new ShoppingCart(calc) { Products = products };
    decimal totalValue = cart.CalculateProductTotal();
    return View(totalValue);
}
...

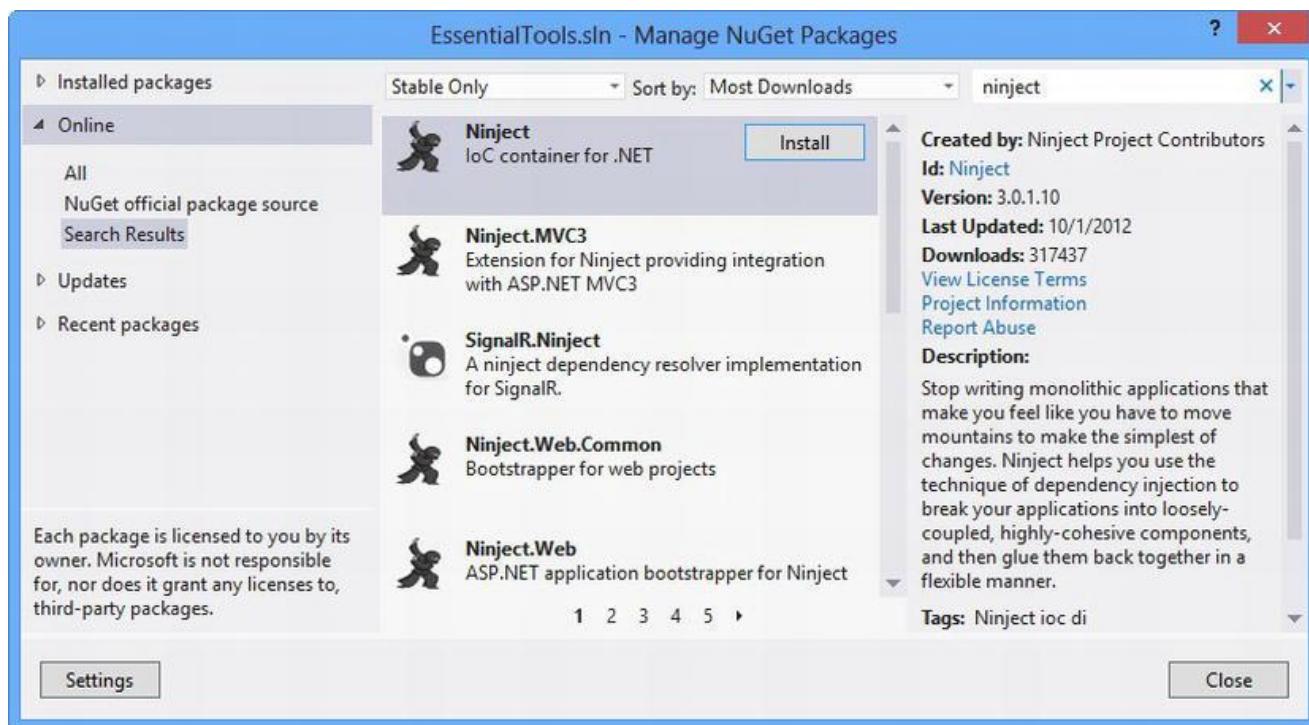
```

Мы хотим использовать Ninject, чтобы достичь того этапа, когда мы указываем, что мы хотим создать экземпляр реализации интерфейса `IValueCalculator`, но детали о том, какая требуется реализация, не являлись бы частью кода в контроллере `Home`.

## Добавление Ninject в проект Visual Studio

Самый простой способ добавить Ninject в MVC проект – это использовать интегрированную поддержку Visual Studio для `NuGet`, что облегчает установку большого набора пакетов и поддерживает их в актуальном состоянии. Выберите `Tools Library Package Manager Manage NuGet Packages for Solution`, чтобы открыть диалоговое NuGet пакетов. Нажмите **Online** на левой панели и введите **Ninject** в поле поиска в правом верхнем углу диалогового окна. Вы увидите ряд пакетов Ninject, похожих на те, что показаны на [рисунке 6-2](#). (Вы можете увидеть другие результаты поиска, которые отображают обновления, выпущенные после того, как мы написали эту главу).

**Рисунок 6-2:** Выбор Ninject из пакетов NuGet



Нажмите кнопку `Install` для библиотеки `Ninject`, и Visual Studio загрузит библиотеку и установит ее в вашем проекте: вы увидите, как `Ninject` появится в разделе `References` проекта.

`NuGet` является отличным инструментом для получения и поддержания пакетов, используемых в проектах Visual Studio, и мы рекомендуем вам использовать его. В этой книге, однако, нам нужно применять другой подход, потому что использование `NuGet` добавляет в проект много логов и дополнительной информации, чтобы была возможность синхронизировать и обновлять пакеты. Эти

дополнительные данные удвоили бы размер нашего простого проекта. И поэтому исходной код, доступный для скачивания на [Apress.com](http://Apress.com), стал бы слишком большим для многих читателей.

Вместо этого мы загрузили последнюю версию библиотеки с веб сайта Ninject ([www.ninject.org](http://www.ninject.org)) и установили его вручную, выбрав Add Reference из меню Project в Visual Studio, нажали на кнопку **Browse**, перешли к файлу и распаковали zip файл Ninject. Мы выбрали файл Ninject.dll и добавили его в проект вручную. Мы получили такой же результат, как если бы использовали NuGet, но, конечно, мы не сможем получить выгоду от простого обновления и управления пакетами.

#### Примечание

Внесем ясность: единственная причина, почему мы добавили библиотеку Ninject вручную, заключается в том, чтобы сохранить довольно небольшой размер исходного кода, прилагаемого к этой книге. В наших собственных проектах, где несколько дополнительных мегабайт данных не имеют никакого значения, мы используем NuGet. Мы рекомендуем вам также использовать NuGet.

## Приступим к работе с Ninject

Есть три стадии начала работы с основным функционалом Ninject, и все они представлены в листинге 6-9. Тут выделены изменения, которые мы внесли в контроллер Home.

### Листинг 6-9: Добавление базового функционала Ninject в метод действия Index

```
using EssentialTools.Models;
using System.Web.Mvc;
using System.Linq;
using Ninject;
namespace EssentialTools.Controllers
{
    public class HomeController : Controller
    {
        private Product[] products =
        {
            new Product {Name = "Kayak", Category = "Watersports", Price = 275M},
            new Product {Name = "Lifejacket", Category = "Watersports", Price = 48.95M},
            new Product {Name = "Soccer ball", Category = "Soccer", Price = 19.50M},
            new Product {Name = "Corner flag", Category = "Soccer", Price = 34.95M}
        };
        public ActionResult Index()
        {
            IKernel ninjectKernel = new StandardKernel();
            ninjectKernel.Bind<IValueCalculator>().To<LinqValueCalculator>();
            IValueCalculator calc = ninjectKernel.Get<IValueCalculator>();
            ShoppingCart cart = new ShoppingCart(calc) { Products = products };
            decimal totalValue = cart.CalculateProductTotal();
            return View(totalValue);
        }
    }
}
```

Первый этап заключается в подготовке Ninject для использования. Нам нужно создать экземпляр Ninject ядра – это объект, который мы будем использовать для связи с Ninject и запросом реализаций интерфейса. Вот выражение из листинга, которое создает ядро:

```
...
IKernel ninjectKernel = new StandardKernel();
...
```

Нам нужно создать реализацию интерфейса `Ninject.IKernel`, что мы делаем, создав новый экземпляр класса `StandardKernel`. Это позволит нам выполнить второй этап, который заключается в создании связи между интерфейсами в нашем приложении и реализациями классов, с которыми мы хотим работать. Вот выражение из листинга, которое это делает:

```
...
ninjectKernel.Bind<IValueCalculator>().To<LinqValueCalculator>();
...
```

`Ninject` использует параметры C# типов, чтобы создать связь: мы устанавливаем интерфейс, с которым мы хотим работать, в качестве параметра типа для метода `Bind` и вызываем метод `To` для результата, который он возвращает. Мы устанавливаем класс реализации, для которого мы хотим создать экземпляр, в качестве параметра типа для метода `To`. Это выражение говорит `Ninject`, что когда его попросят реализовать интерфейс `IValueCalculator`, он должен выполнить запрос, создав новый экземпляр класса `LinqValueCalculator`.

Последний этап заключается в реальном использовании `Ninject`, что мы делаем при помощи метода `Get`:

```
...
IValueCalculator calc = ninjectKernel.Get<IValueCalculator>();
...
```

Параметр типа, используемый методом `Get`, говорит `Ninject`, в каком интерфейсе мы заинтересованы, а результаты этого метода являются экземпляром типа реализации, который мы только что определили для метода `To`.

## Установка MVC DI

Результатом этих трех этапов, которые мы показали вам в предыдущем листинге, является то, что мы знаем, какой класс реализации должен быть использован для выполнения запросов для интерфейса `IValueCalculator`, который был установлен в `Ninject`. Но, конечно, мы еще не улучшили наше приложение, потому что все это по-прежнему определяется в контроллере `Home`: это означает, что контроллер `Home` по-прежнему тесно связан с классом `LinqValueCalculator`.

В следующих разделах мы покажем вам, как вставить `Ninject` в сердце MVC приложения, что позволит нам упростить контроллер, расширить влияние `Ninject`, чтобы он работал по всему приложению и использовал свои дополнительные возможности.

### Создание DR (Dependency Resolver)

Первое изменение, которое мы собираемся сделать, заключается в создании DR (*dependency resolver*: функциональная возможность, которая отвечает за создание зависимостей). MVC использует DR для создания экземпляров классов, для которых он должен обрабатывать запросы. Создавая DR, мы гарантируем, что `Ninject` будет использоваться всякий раз, когда будет создан объект.

Добавьте в наш проект новую папку `Infrastructure` и добавьте новый файл класса `NinjectDependencyResolver.cs`. Убедитесь, что содержимое этого файла соответствует [листингу 6-10](#).

#### Листинг 6-10: Содержимое файла `NinjectDependencyResolver.cs`

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

```

using Ninject;
using Ninject.Parameters;
using Ninject.Syntax;
using System.Configuration;
using EssentialTools.Models;
namespace EssentialTools.Infrastructure
{
    public class NinjectDependencyResolver : IDependencyResolver
    {
        private IKernel kernel;
        public NinjectDependencyResolver()
        {
            kernel = new StandardKernel();
            AddBindings();
        }
        public object GetService(Type serviceType)
        {
            return kernel.TryGet(serviceType);
        }
        public IEnumerable<object> GetServices(Type serviceType)
        {
            return kernel.GetAll(serviceType);
        }
        private void AddBindings()
        {
            kernel.Bind<IValueCalculator>().To<LinqValueCalculator>();
        }
    }
}

```

MVC фреймворк вызовет методы `.GetService` или `GetServices`, когда ему будет нужен экземпляр класса для обработки входящих запросов. Работа DR заключается в создании этого экземпляра: задача, которую мы выполняем при помощи методов `Ninject TryGet` и  `GetAll`. Метод `TryGet` работает как метод `Get`, который мы использовали ранее, но он возвращает `null`, если нет подходящей связки, а не выбрасывает исключение. Метод  `GetAll` поддерживает несколько связок для одного типа.

Наш класс DR также находится там, где мы установили наши `Ninject` связки. В методе `AddBindings` мы используем методы `Bind` и `To`, чтобы установить связь между интерфейсом `IValueCalculator` и классом `LinqValueCalculator`.

## Регистрация DR

Нам нужно сказать MVC, что мы хотим использовать наш собственный DR, мы это можем сделать, изменив файл `Global.asax.cs`. Вы можете увидеть дополнения, которые мы внесли в класс `MVCAplication`, содержащийся в этом файле, в [листинге 6-11](#).

### Листинг 6-11: Регистрация DR

```

using EssentialTools.Infrastructure;
using System.Web.Http;
using System.Web.Mvc;
using System.Web.Routing;
namespace EssentialTools
{
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();
            DependencyResolver.SetResolver(new NinjectDependencyResolver());
        }
    }
}

```

```

        WebApiConfig.Register(GlobalConfiguration.Configuration);
        FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
        RouteConfig.RegisterRoutes(RouteTable.Routes);
    }
}
}

```

С этим дополнением Ninject будет предложена возможность создать любой экземпляр объекта, который требует MVC фреймворк, разместив DI прямо в ядро нашего MVC приложения.

## Рефакторинг контроллера Home

Последним шагом является рефакторинг контроллера `Home`, чтобы он использовал возможности, которые мы создали в предыдущих разделах. Изменения, которые мы сделали, представлены в [листе 6-12](#).

### Листинг 6-12: Рефакторинг контроллера Home

```

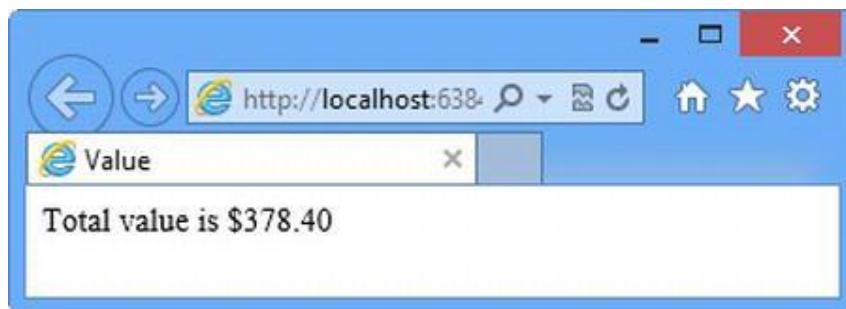
using EssentialTools.Models;
using System.Web.Mvc;
using System.Linq;
namespace EssentialTools.Controllers
{
    public class HomeController : Controller
    {
        private Product[] products =
        {
            new Product {Name = "Kayak", Category = "Watersports", Price = 275M},
            new Product {Name = "Lifejacket", Category = "Watersports", Price = 48.95M},
            new Product {Name = "Soccer ball", Category = "Soccer", Price = 19.50M},
            new Product {Name = "Corner flag", Category = "Soccer", Price = 34.95M}
        };
        private IValueCalculator calc;
        public HomeController(IValueCalculator calcParam)
        {
            calc = calcParam;
        }
        public ActionResult Index()
        {
            ShoppingCart cart = new ShoppingCart(calc) { Products = products };
            decimal totalValue = cart.CalculateProductTotal();
            return View(totalValue);
        }
    }
}

```

Основное изменение, которое мы здесь сделали, заключается в том, что мы добавили конструктор, который принимает реализацию интерфейса `IValueCalculator`. Мы не указали, с какой реализацией мы хотим работать, и мы добавили переменную экземпляра `calc`, которую мы можем использовать для обращения к `IValueCalculator`, который мы получаем в конструктор через класс контроллера.

Другое изменение, которые мы сделали, заключается в том, чтобы удалить любые упоминания о Ninject коде или классе `LinqValueCalculator`: наконец, мы разбили сильную связь между `HomeController` и классом `LinqValueCalculator`. Если вы запустите пример, вы увидите результат, показанный на [рисунке 6-3](#). Естественно, мы получили тот же результат, что и когда мы создавали экземпляр класса `LinqValueCalculator` непосредственно в контроллере.

**Рисунок 6-3:** Результат запуска приложения



То, что мы создали, является примером *внедрения через конструктор*, это одна из форм внедрения зависимостей. Вот то, что произошло, когда вы запустили пример приложения и Internet Explorer сделал запрос на корневой URL приложения:

1. MVC фреймворк получил запрос и выяснил, что запрос предназначен для контроллера `Home` (мы объясним, как MVC фреймворк выясняет это, в главе 15).
2. MVC фреймворк попросил наш пользовательский DR класс создать новый экземпляр класса `HomeController`, указав класс, используя параметр `Type` метода `GetService`.
3. Наш DR попросил Ninject создать новый класс `HomeController` при помощи передачи объекта `Type` методу `TryGet`.
4. Ninject изучил конструктор `HomeController` и обнаружил, что он требует реализацию `IValueCalculator`.
5. Ninject создает экземпляр класса `LinqValueCalculator` и использует его для создания нового экземпляра класса `HomeController`.
6. Ninject передает вновь созданный экземпляр `HomeController` пользовательскому DR, который возвращает его MVC фреймворку. MVC фреймворк использует экземпляр контроллера для обработки запроса.

Мы все это объяснили, потому что DI может показаться немного запутанным, если вы впервые с ним сталкиваетесь. Одно из преимуществ такого подхода заключается в том, что любой контроллер может объявить, что он требует в своем конструкторе `IValueCalculator` и что будет использоваться Ninject. Наш пользовательский DR создает экземпляр реализации, которую мы указали в методе `AddBindings`.

Самое приятное то, что нам нужно изменить только наш класс DR, если мы хотим заменить `LinqValueCalculator` другой реализацией, потому что это единственное место, где требуется определить реализацию, которая используется для выполнения запросов для интерфейса `IValueCalculator`.

## Создание цепочек зависимостей

Когда вы просите Ninject создать тип, он проверяет связи между этим типом и другими типами. Если есть дополнительные зависимости, Ninject автоматически схватывает их и создает экземпляры всех требуемых классов. Чтобы показать эту функцию, мы добавили файл `Discount.cs` в папку `Models` проекта и определили новый интерфейс и класс, реализующий его, как представлено в листинге 6-13.

**Листинг 6-13:** Определение нового интерфейса и реализации

```
namespace EssentialTools.Models
{
    public interface IDiscountHelper
    {
        decimal ApplyDiscount(decimal totalParam);
```

```

    }
    public class DefaultDiscountHelper : IDiscountHelper
    {
        public decimal ApplyDiscount(decimal totalParam)
        {
            return (totalParam - (10m / 100m * totalParam));
        }
    }
}

```

`IDiscountHelper` определяет метод `ApplyDiscount`, который будет применять скидку к значению `decimal`. Класс `DefaultDiscountHelper` реализует интерфейс и применяет фиксированную 10-процентную скидку. Мы изменили класс `LinqValueCalculator`, так что он использует интерфейс `IDiscountHelper` при выполнении расчетов, как показано в листинге 6-14.

#### Листинг 6-14: Добавление зависимости в класс `LinqValueCalculator`

```

using System.Collections.Generic;
using System.Linq;
namespace EssentialTools.Models
{
    public class LinqValueCalculator : IValueCalculator
    {
        private IDiscountHelper discounter;
        public LinqValueCalculator(IDiscountHelper discountParam)
        {
            discounter = discountParam;
        }
        public decimal ValueProducts(IEnumerable<Product> products)
        {
            return discounter.ApplyDiscount(products.Sum(p => p.Price));
        }
    }
}

```

Вновь добавленный конструктор класса принимает реализацию интерфейса `IDiscountHelper`, которая затем используется в методе `ValueProducts`, чтобы применить скидку к совокупной стоимости обрабатываемых объектов `Product`.

Мы связываем интерфейс `IDiscountHelper` с классом реализации, используя ядро `Ninject` в классе `NinjectDependencyResolver`, так же как мы сделали для `IValueCalculator`, как показано в листинге 6-15.

#### Листинг 6-15: Связывание другого интерфейса с его реализацией

```

using System;
using System.Collections.Generic;
using System.Web.Mvc;
using Ninject;
using Ninject.Parameters;
using Ninject.Syntax;
using System.Configuration;
using EssentialTools.Models;
namespace EssentialTools.Infrastructure
{
    public class NinjectDependencyResolver : IDependencyResolver
    {
        private IKernel kernel;
        public NinjectDependencyResolver()
        {
            kernel = new StandardKernel();
            AddBindings();
        }
}

```

```

    }
    public object GetService(Type serviceType)
    {
        return kernel.TryGet(serviceType);
    }
    public IEnumerable<object> GetServices(Type serviceType)
    {
        return kernel.GetAll(serviceType);
    }
    private void AddBindings()
    {
        kernel.Bind<IValueCalculator>().To<LinqValueCalculator>();
        kernel.Bind<IDiscountHelper>().To<DefaultDiscountHelper>();
    }
}
}

```

Мы создали цепочку зависимостей, которую легко обрабатывает Ninject, используя связи, которые мы определили в пользовательском DR. Для того чтобы удовлетворить запрос для класса `HomeController`, Ninject считает, что необходимо создать реализацию для класса `IValueCalculator`, и он делает это, рассматривая свои связи и понимая, что наша политика для этого интерфейса заключается в использовании класса `LinqValueCalculator`. Но для того, чтобы создать объект `LinqValueCalculator`, Ninject понимает, что он должен использовать реализацию `IDiscountHelper`, и поэтому он рассматривает связи и создает объект `DefaultDiscountHelper`. Он создает `DefaultDiscountHelper` и передает его конструктору объекта `LinqValueCalculator`, который в свою очередь передает его конструктору класса `HomeController`. Затем он используется для обслуживания запроса пользователя. Ninject проверяет каждый класс, для которого он создал экземпляр для внедрения зависимостей, таким образом неважно, насколько длинными и сложными являются цепочки зависимостей.

## Определение значений свойств и параметров конструктора

Мы можем конфигурировать классы, которые создает Ninject, путем предоставления информации о нужных значениях, применяемых к свойствам, когда мы связываем интерфейс с его реализацией. Чтобы продемонстрировать эту функцию, мы изменили класс `DefaultDiscountHelper` так, что он определяет свойство `DiscountSize`, которое используется для расчета суммы скидки, как показано в листинге 6-16.

**Листинг 6-16:** Добавление свойства в класс реализации

```

namespace EssentialTools.Models
{
    public interface IDiscountHelper
    {
        decimal ApplyDiscount(decimal totalParam);
    }
    public class DefaultDiscountHelper : IDiscountHelper
    {
        public decimal DiscountSize { get; set; }
        public decimal ApplyDiscount(decimal totalParam)
        {
            return (totalParam - (DiscountSize / 100m * totalParam));
        }
    }
}

```

Когда мы привязываем конкретный класс к типу при помощи Ninject, мы можем использовать метод `WithPropertyValue` для установки значения свойства `DiscountSize` в классе

`DefaultDiscountHelper`. Вы можете увидеть изменение, которое мы внесли в метод `AddBindings` класса `NinjectDependencyResolver`, в листинге 6-17.

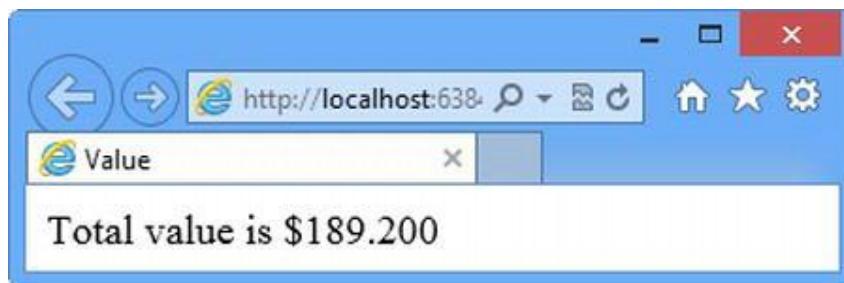
**Листинг 6-17:** Использование `Ninject` метода `WithPropertyValue`

```
private void AddBindings() {
    kernel.Bind<IValueCalculator>().To<LinqValueCalculator>();
    kernel.Bind<IDiscountHelper>()
        .To<DefaultDiscountHelper>().WithPropertyValue("DiscountSize", 50M);
}
```

Обратите внимание, что мы должны указать имя свойства в виде строки. Нам не нужно менять любую другую связь, а также менять то, как мы используем метод `Get` для получения экземпляра класса `ShoppingCart`.

Значение свойства установлено в связке с классом `DefaultDiscountHelper`, и его результат заключается в вычислении общей стоимости предметов со скидкой. Это изменение показано на рисунке 6-4.

**Рисунок 6-4:** Результат применения скидки через свойство при работе с цепочкой зависимостей



Если вам нужно установить более одного значения свойства, можно составить цепочку вызовов метода `WithPropertyValue` и охватить их всех. Мы можем сделать то же самое с параметрами конструктора. В листинге 6-18 показан класс `DefaultDiscounterHelper`, переделанный таким образом, чтобы размер скидки передавался в качестве параметра конструктора.

**Листинг 6-18:** Использование свойства конструктора в классе реализации

```
namespace EssentialTools.Models
{
    public interface IDiscountHelper
    {
        decimal ApplyDiscount(decimal totalParam);
    }
    public class DefaultDiscountHelper : IDiscountHelper
    {
        public decimal discountSize;
        public DefaultDiscountHelper(decimal discountParam)
        {
            discountSize = discountParam;
        }
        public decimal ApplyDiscount(decimal totalParam)
        {
            return (totalParam - (discountSize / 100m * totalParam));
        }
    }
}
```

Чтобы связать этот класс, используя `Ninject`, мы указываем значение параметра конструктора при помощи метода `WithConstructorArgument` в методе `AddBindings`, как показано в листинге 6-19.

### Листинг 6-19: Связывание с классом, который требует параметр конструктора

```
private void AddBindings() {
    kernel.Bind<IValueCalculator>().To<LinqValueCalculator>();
    kernel.Bind<IDiscountHelper>()
        .To<DefaultDiscountHelper>().WithConstructorArgument("discountParam", 50M);
}
```

Эта техника позволяет внедрять значение в конструктор. Еще раз, можно связать вызовы этих методов, чтобы указать несколько значений, а затем смешивать и сочетать их с зависимостями. Ninject выяснит, что нам нужно, и создаст это соответствующим образом.

#### Совет

*Обратите внимание, что мы не просто изменили вызов WithPropertyValue к WithConstructorArgument. Мы также изменили имя нужного члена, чтобы он соответствовал соглашению C# по именам параметров.*

## Использование условной связки

Ninject поддерживает ряд условных связывающих методов, которые позволяют нам указать, какие классы должны использоваться для ответов на запросы для конкретных запросов. Чтобы продемонстрировать эту функцию, мы добавили новый файл FlexibleDiscountHelper.cs в папку Models нашего проекта, содержимое которого вы можете увидеть в листинге 6-20.

### Листинг 6-20: Содержание файла FlexibleDiscountHelper.cs

```
namespace EssentialTools.Models
{
    public class FlexibleDiscountHelper : IDiscountHelper
    {
        public decimal ApplyDiscount(decimal totalParam)
        {
            decimal discount = totalParam > 100 ? 70 : 25;
            return (totalParam - (discount / 100m * totalParam));
        }
    }
}
```

Класс FlexibleDiscountHelper применяет различные скидки, исходя из величины общей суммы, на которую распространяется скидка. Затем мы можем изменить метод AddBindings в NinjectDependencyResolver, чтобы сказать Ninject, когда мы хотим использовать FlexibleDiscountHelper и когда мы хотим использовать DefaultDiscountHelper, как показано в листинге 6-21.

### Листинг 6-21: Обновление метода AddBindings для использования условной связки

```
private void AddBindings() {
    kernel.Bind<IValueCalculator>().To<LinqValueCalculator>();
    kernel.Bind<IDiscountHelper>()
        .To<DefaultDiscountHelper>().WithConstructorArgument("discountParam", 50M);
    kernel.Bind<IDiscountHelper>().To<FlexibleDiscountHelper>()
        .WhenInjectedInto<LinqValueCalculator>();
}
```

Новая связка указывает, что если класс FlexibleDiscountHelper должен использоваться в качестве реализации интерфейса IDiscountHelper, тогда Ninject будет внедрять реализацию в объект LinqValueCalculator.

Мы оставили на месте исходную связку для `IDiscountHelper`. Ninject пытается найти наиболее подходящее совпадение, и поэтому стоит оставлять связку по умолчанию для того же класса или интерфейса, так чтобы у Ninject был резервный вариант, если критерии для условной связки не могут быть удовлетворены. Ninject поддерживает несколько различных условных связующих методов, и наиболее полезные из них мы привели в таблице 6-1.

**Таблица 6-1:** Условные связующие методы Ninject

Метод	Результат
<code>When(predicate)</code>	Связка используется, если утверждение (predicate) – лямбда-выражение – считается верным.
<code>WhenClassHas&lt;T&gt;()</code>	Связка используется, если внедряемый класс, аннотируется атрибутом, чей тип определяется <code>T</code> .
<code>WhenInjectedInto&lt;T&gt;()</code>	Связка используется, если внедряемый класс принадлежит к типу <code>T</code> .

## Модульное тестирование при помощи Visual Studio

Есть много .NET пакетов для модульного тестирования, многие из которых имеют открытый исходный код и находятся в свободном доступе. В этой книге мы будем использовать встроенную поддержку модульного тестирования, которая поставляется с Visual Studio, но есть и другие доступные .NET пакеты для модульного тестирования. Наиболее популярным является, вероятно, NUnit, но все остальные пакеты для юнит тестирования делают фактически то же самое. Причина, почему мы выбрали поддержку юнит тестирования Visual Studio, заключается в том, что нам нравится интеграция с остальной частью IDE. Хотя с Visual Studio 2012 Microsoft дал возможность интегрировать сторонние библиотеки тестирования в IDE, чтобы вы могли работать с ними так же, как и со встроенными инструментами тестирования.

Чтобы показать поддержку модульного тестирования Visual Studio, мы собираемся добавить новую реализацию интерфейса `IDiscountHelper` в проект для примера. Создайте новый файл `MinimumDiscountHelper.cs` в папке `Models`. Убедитесь, что содержание соответствует показанному в листинге 6-22.

**Листинг 6-22:** Содержание файла `MinimumDiscountHelper.cs`

```
using System;
namespace EssentialTools.Models
{
    public class MinimumDiscountHelper : IDiscountHelper
    {
        public decimal ApplyDiscount(decimal totalParam)
        {
            throw new NotImplementedException();
        }
    }
}
```

Наша цель в данном примере заключается в том, чтобы `MinimumDiscountHelper` показал следующее поведение:

- Если общая сумма превышает \$100, скидка составит 10 процентов.
- Если общая сумма составляет от \$10 до \$100 включительно, скидка будет \$5.
- Для общей суммы меньше \$10 скидки не будет.

- Если общая сумма является отрицательным числом, выбрасывается исключение `ArgumentOutOfRangeException`.

Наш класс `MinimumDiscountHelper` не реализует пока еще ни одной из этих форм поведения: мы собираемся следовать подходу Test Driven Development (TDD), когда сначала пишутся юнит тесты, а только затем создается код.

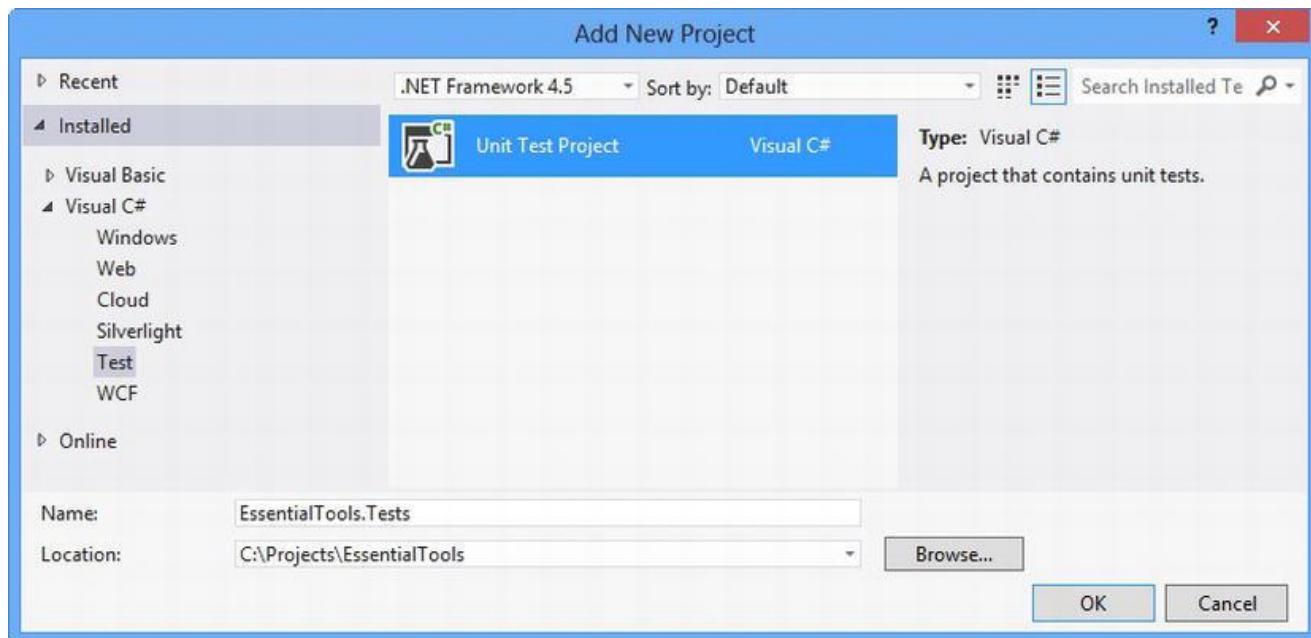
## Создание проекта по юнит тестированию

Первый шаг, который нам нужно сделать, это создать проект для модульного тестирования. Щелкните правой кнопкой мыши по Solution 'EssentialTools' нашего проекта в Solution Explorer и выберите Add New Project из всплывающего меню.

### *Совет*

*Вы можете создать проект по тестированию, когда вы создаете новый MVC проект: в диалоговом окне, где вы выбираете начальный контент для MVC проекта, есть опция Create a unit test project.*

Рисунок 6-5: Создание юнит тест проекта

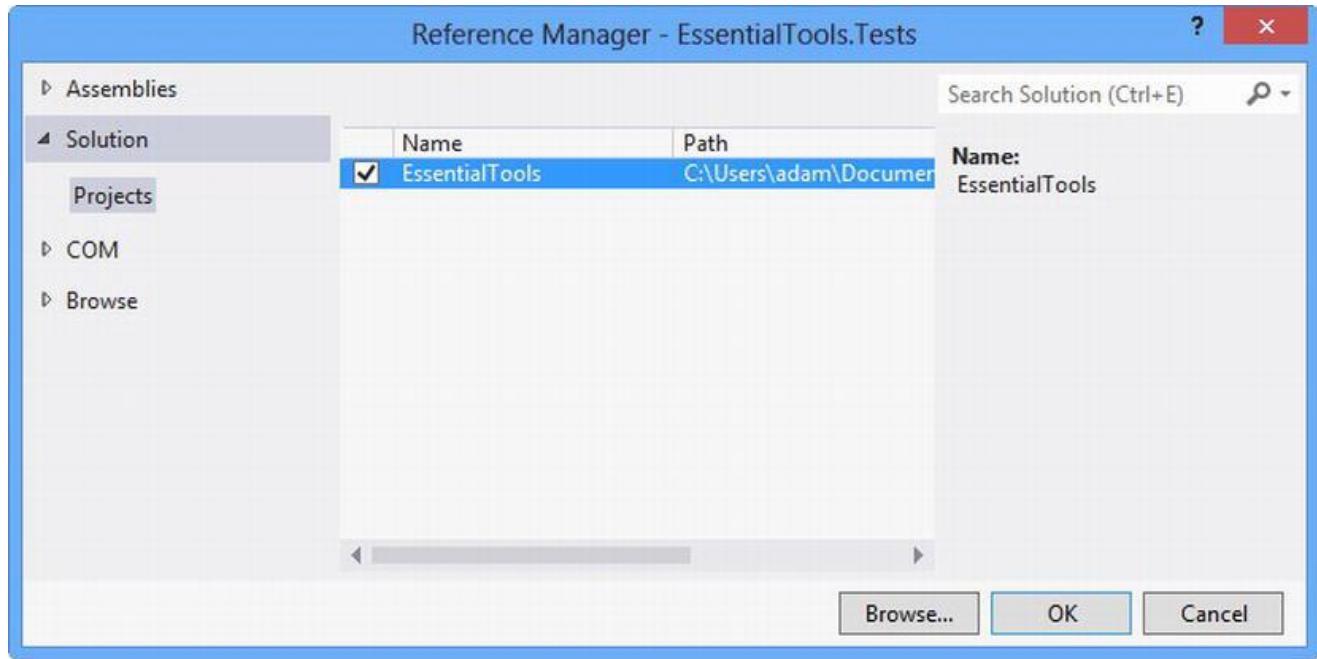


Назовите проект `EssentialTools.Tests` и нажмите кнопку OK, чтобы создать новый проект, который будет добавлен к текущему решению Visual Studio вместе с проектом MVC приложения.

Нам нужно добавить ссылку на тестовый проект, чтобы мы могли использовать его для выполнения тестов по классам MVC проекта. Щелкните правой кнопкой мыши в Solution Explorer по References для проекта `EssentialTools.Tests`, а затем выберите Add Reference из всплывающего меню.

Нажмите Solution в левой панели и поставьте рядом с `EssentialTools`, как показано на рисунке 6-6.

**Рисунок 6-6:** Добавление ссылки в MVC проект



## Создание юнит тестов

Мы добавим наши юнит тесты в файл `UnitTest1.cs` проекта `EssentialTools.Tests`. В платных выпусках Visual Studio есть некоторые полезные функции для автоматической генерации тестовых методов, которые не доступны в Visual Studio Express, но мы все же можем создавать полезные и хорошие тесты (и наш опыт по автоматически генерируемым тестам был довольно успешный). Чтобы начать работу, мы внесли изменения, показанные в листинге 6-23.

**Листинг 6-23:** Добавление тестовых методов в файл `UnitTest1.cs`

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using EssentialTools.Models;
namespace EssentialTools.Tests
{
    [TestClass]
    public class UnitTest1
    {
        private IDiscountHelper getTestObject()
        {
            return new MinimumDiscountHelper();
        }
        [TestMethod]
        public void Discount_Above_100()
        {
            // arrange
            IDiscountHelper target = getTestObject();
            decimal total = 200;
            // act
            var discountedTotal = target.ApplyDiscount(total);
            // assert
            Assert.AreEqual(total * 0.9M, discountedTotal);
        }
    }
}
```

Мы только что добавили один юнит тест. Класс, который содержит тесты, помечается атрибутом `TestClass`, а отдельными тестами являются методы, отмеченные атрибутом `TestMethod`. Не все методы в классе модульных тестов должны быть юнит тестами. Чтобы показать это, мы определили метод `getTestObject`, который мы будем использовать, чтобы регулировать наши тесты. Поскольку этот метод не имеет атрибута `TestMethod`, Visual Studio не будет обрабатывать его как юнит тест.

#### *Совет*

*Обратите внимание, что мы должны были добавить выражение `using`, чтобы импортировать пространство имен `EssentialTools.Models` в тестовый класс.*

*Тестовые классы не отличаются от обычных C# классов: только атрибуты `TestClass` и `TestMethod` добавляют магию тестирования в проект.*

Вы видите, что мы следовали паттерну *arrange/act/assert* (A/A/A) в методе модульного теста. Есть бесчисленное множество соглашений о том, как называть юнит тесты, но мы считаем, что нужно просто использовать имена, которые дают понять, что проверяет тест. Наш метод модульного теста называется `Discount_Above_100`, что является для нас понятным и имеющим смысл. Но на самом деле, важно, чтобы вы (и ваша команда) понимали, на каком соглашении стоит остановиться, если вы хотите принять другую схему имен, отличную от нашей.

Сперва мы вызвали метод `getTestObject`. Он создает экземпляр объекта, который мы собираемся тестировать: в данном случае это класса `MinimumDiscountHelper`. Мы также определяем значение `total`, которое мы собираемся протестировать. Это часть *arrange* нашего модульного теста.

Для части *act* нашего теста мы вызываем метод `MinimumDiscountHelper.ApplyDiscount` и присваиваем результат переменной `discountedTotal`. Наконец, для части *assert* теста мы используем метод `Assert.AreEqual`, чтобы проверить, что значение, которое мы получили от метода `ApplyDiscount`, составляет 90% от первоначальной общей стоимости.

Класс `Assert` имеет ряд статических методов, которые вы можете использовать в своих тестах. Этот класс можно найти в пространстве имен `Microsoft.VisualStudio.TestTools.UnitTesting` наряду с некоторыми дополнительными классами, которые могут быть полезны для создания и проведения тестов. Вы можете узнать больше о классах в данном пространстве имен на <http://msdn.microsoft.com/en-us/library/ms182530.aspx>.

Мы наиболее часто используем класс `Assert`, и мы привели наиболее важные методы в таблице 6-2 (хотя их гораздо больше, и их стоит изучить).

**Таблица 6-2:** Статические методы класса `Assert`

Метод	Описание
<code>AreEqual&lt;T&gt;(T, T), AreEqual&lt;T&gt;(T, T, string)</code>	Утверждает, что два объекта типа <code>T</code> имеют одинаковое значение.
<code>AreNotEqual&lt;T&gt;(T, T), AreNotEqual&lt;T&gt;(T, T, string)</code>	Утверждает, что два объекта типа <code>T</code> не имеют одинакового значения.
<code>AreSame&lt;T&gt;(T, T), AreSame&lt;T&gt;(T, T, string)</code>	Утверждает, что две переменные относятся к одному объекту.
<code>AreNotSame&lt;T&gt;(T, T), AreNotSame&lt;T&gt;(T, T, string)</code>	Утверждает, что две переменные относятся к разным объектам.
<code>Fail(), Fail(string)</code>	Утверждение не выполнилось: условия не проверены.

Метод	Описание
Inconclusive(), Inconclusive(string)	Указывает, что результат модульного теста не может быть окончательно установлен.
IsTrue(bool), IsTrue(bool, string)	Утверждает, что значение bool верно: чаще всего используется для оценки выражения, возвращающего булев результат.
IsFalse(bool), IsFalse(bool, string)	Утверждает, что значение bool ложно.
IsNull(object), IsNull(object, string)	Утверждает, что переменной не присвоена ссылка на объект.
IsNotNull(object), IsNotNull(object, string)	Утверждает, что переменной присвоена ссылка на объект.
IsInstanceOfType(object, Type), IsInstanceOfType(object, Type, string)	Утверждает, что это объект указанного типа или унаследован от указанного типа.
IsNotInstanceOfType(object, Type), IsNotInstanceOfType(object, Type, string)	Утверждает, что этот объект не является объектом указанного типа.

Каждый из статических методов класса `Assert` позволяет проверить некоторые аспекты вашего модульного теста. Если утверждение не выполняется, выбрасывается исключение, это обозначает, что весь тест не сработал. Каждый модульный тест рассматривается отдельно, так что другие тесты будут продолжать выполняться.

Каждый из этих методов имеет перегруженную версию, которая принимает параметр `string`. Стока включена в качестве сообщения об исключении, если утверждение не выполняется. Методы `AreEqual` и `AreNotEqual` имеют ряд перегруженных версий, которые предназначены для сопоставления конкретных типов. Например, есть версия, которая позволяет сравнивать строки, не принимая во внимание регистр.

#### Совет

*Одним из примечательных членов пространства имен*

*Microsoft.VisualStudio.TestTools.UnitTesting* является атрибут `ExpectedException`.

*Это утверждение, которое будет выполнено только в том случае, если модульный тест выбросит исключение типа, указанного параметром `ExceptionType`. Это аккуратный способ ловить исключения без необходимости возиться с блоками `try...catch` в юнит teste.*

После того как мы показали вам, как собрать вместе юнит тесты, мы добавили тесты в проект для проверки других видов поведения, которые мы описали для нашего `MinimumDiscountHelper`. Вы можете увидеть дополнения в листинге 6-24, но эти модульные тесты настолько короткие и простые (что, как правило, характерно для юнит тестов), что мы не собираемся детально их объяснять.

#### Листинг 6-24: Определение оставшихся тестов

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using EssentialTools.Models;
namespace EssentialTools.Tests
{
    [TestClass]
    public class UnitTest1
    {
        private IDiscountHelper getTestObject()
        {
            return new MinimumDiscountHelper();
```

```

    }

    [TestMethod]
    public void Discount_Above_100()
    {
        // arrange
        IDiscountHelper target = getTestObject();
        decimal total = 200;
        // act
        var discountedTotal = target.ApplyDiscount(total);
        // assert
        Assert.AreEqual(total * 0.9M, discountedTotal);
    }

    [TestMethod]
    public void Discount_Between_10_And_100()
    {
        //arrange
        IDiscountHelper target = getTestObject();
        // act
        decimal TenDollarDiscount = target.ApplyDiscount(10);
        decimal HundredDollarDiscount = target.ApplyDiscount(100);
        decimal FiftyDollarDiscount = target.ApplyDiscount(50);
        // assert
        Assert.AreEqual(5, TenDollarDiscount, "$10 discount is wrong");
        Assert.AreEqual(95, HundredDollarDiscount, "$100 discount is wrong");
        Assert.AreEqual(45, FiftyDollarDiscount, "$50 discount is wrong");
    }

    [TestMethod]
    public void Discount_Less_Than_10()
    {
        //arrange
        IDiscountHelper target = getTestObject();
        // act
        decimal discount5 = target.ApplyDiscount(5);
        decimal discount0 = target.ApplyDiscount(0);
        // assert
        Assert.AreEqual(5, discount5);
        Assert.AreEqual(0, discount0);
    }

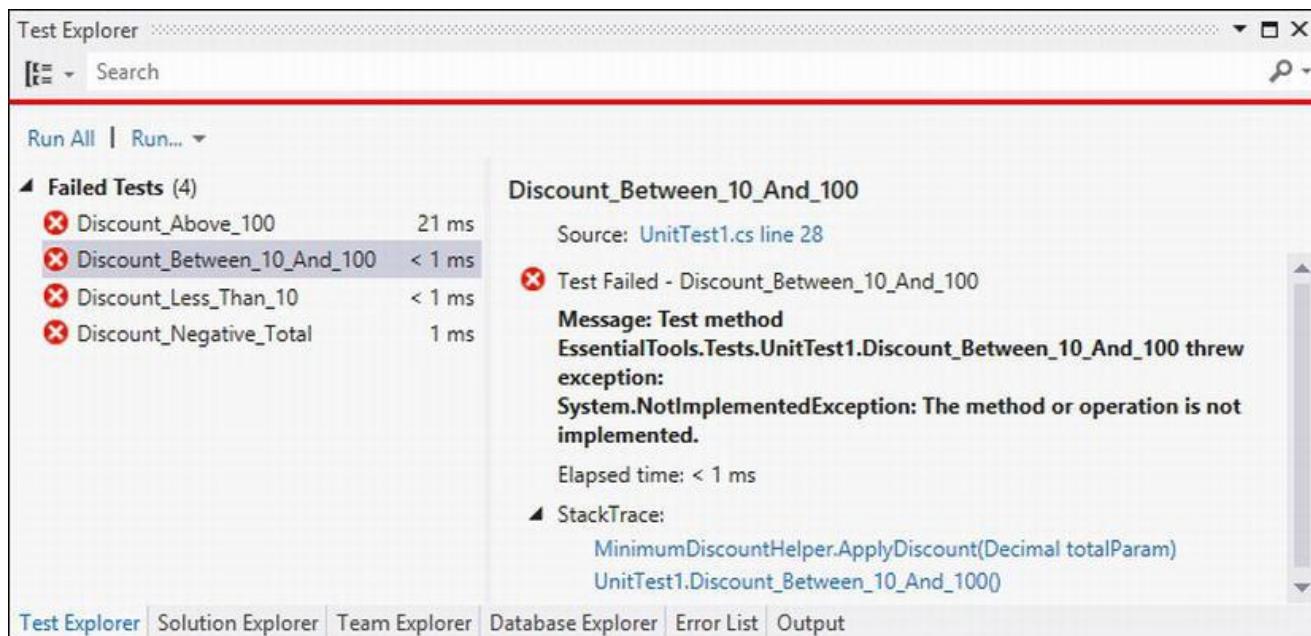
    [TestMethod]
    [ExpectedException(typeof(ArgumentOutOfRangeException))]
    public void Discount_Negative_Total()
    {
        //arrange
        IDiscountHelper target = getTestObject();
        // act
        target.ApplyDiscount(-1);
    }
}
}

```

## **Запуск юнит тестов (и они не срабатывают)**

Visual Studio 2012 представила более полезное окно Test Explorer для управления и запуска тестов. Выберите Windows Test Explorer из меню Test Visual Studio, чтобы увидеть новое окно, и нажмите кнопку Run All в верхнем левом углу. Вы увидите результаты, похожие на те, что показаны на рисунке 6-7.

**Рисунок 6-7:** Запуск тестов для проекта



Вы можете увидеть список тестов, которые мы определили, в левой части окна **Test Explorer**. Естественно, все тесты не сработали, потому что нам еще только предстоит реализовать метод, который мы тестируем. Вы можете нажать на любой из тестов, и в правой части окна вы увидите подробности того, почему тест не сработал. Окно **Test Explorer** предоставляет ряд различных способов выбора и фильтрации юнит тестов, а также тут вы можете запускать отдельные тесты, только те, которые вам надо. Для нашего простого проекта мы все же просто запустим все тесты, нажав на кнопку Run All.

## Реализация тестов

Мы достигли этапа, когда мы можем реализовать функцию, и мы будем знать, что сможем проверить, что код работает, как и ожидается, когда мы закончим. Сейчас мы довольно просто и понятно реализуем класс `MinimumDiscountHelper`, это показано в листинге 6-25.

**Листинг 6-25:** Реализация класса `MinimumDiscountHelper`

```
using System;
namespace EssentialTools.Models
{
    public class MinimumDiscountHelper : IDiscountHelper
    {
        public decimal ApplyDiscount(decimal totalParam)
        {
            if (totalParam < 0)
            {
                throw new ArgumentOutOfRangeException();
            }
            else if (totalParam > 100)
            {
                return totalParam * 0.9M;
            }
            else if (totalParam > 10 && totalParam <= 100)
            {
                return totalParam - 5;
            }
            else
            {

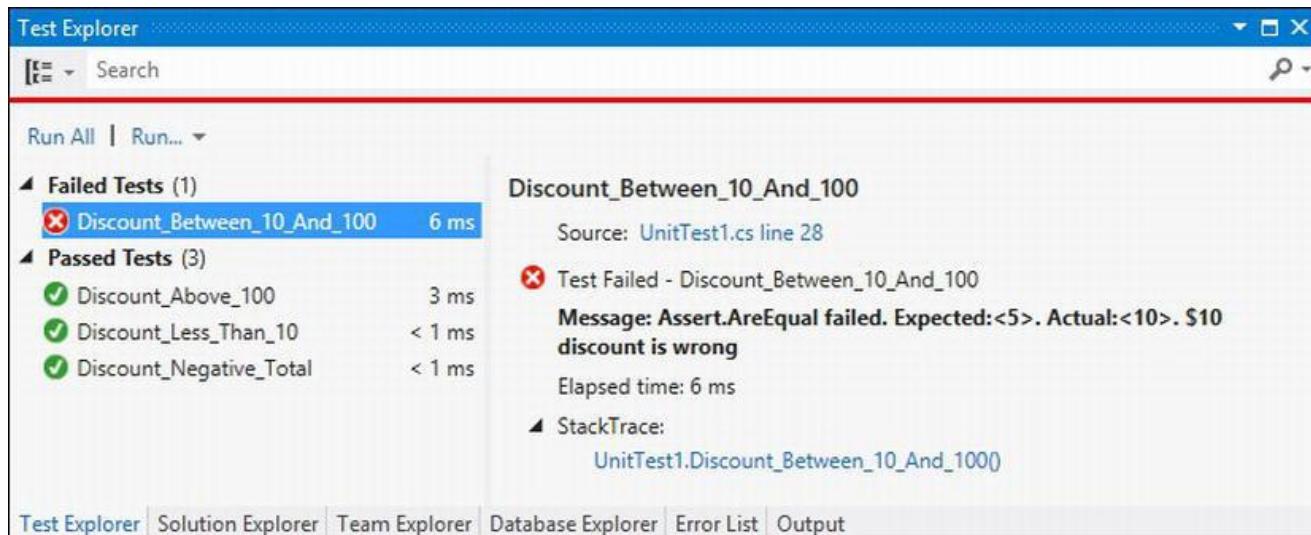
```

```
        return totalParam;  
    }  
}
```

## Тестирование и исправление кода

Мы сознательно оставили ошибку в этом коде, чтобы продемонстрировать, как работает повторяющееся модульное тестирование в Visual Studio. Вы можете увидеть результат ошибки, если нажмете кнопку Run All в окне **Test Explorer**. Результаты тестов показаны на рисунке 6-8.

**Рисунок 6-8:** Ошибка при запуске юнит тестов



Visual Studio всегда пытается дать наиболее полезную информацию в верхней части окна **Test Explorer**. В этой ситуации это означает, что не сработавший тест отображается перед сработавшими.

Вы видите, что три наших модульных теста были успешными, но у нас есть проблема, которая была обнаружена в тестовом методе `Discount_Between_10_And_100`. Если мы нажмем на не сработавший тест, мы увидим, что наш тест ожидал получить результат 5, а на самом деле получил значение 10.

И теперь мы возвращаемся к нашему коду и видим, что мы не совсем правильно реализовали ожидаемое поведение: в частности, скидки для сумм от 10 до 100 не правильно обрабатываются. Проблема лежит в этом выражении класса `MinumumDiscountHelper`:

```
    ...  
} else if (totalParam > 10 && totalParam < 100) {  
    ...
```

В нашей системе скидок мы должны задать поведение для сумм, которые находятся в диапазоне от \$10 до \$100 *включительно*, но наша реализация исключает эти значения и проверяет только те значения, которые больше, чем \$10, а ровно \$10 не проверяет. Решение тут простое, и оно показано в листинге 6-26: должен быть добавлен только один символ, чтобы изменить результат оператора `if`:

### Листинг 6-26: Исправление кода

```
using System;
namespace EssentialTools.Models
{
    public class MinimumDiscountHelper : IDiscountHelper
```

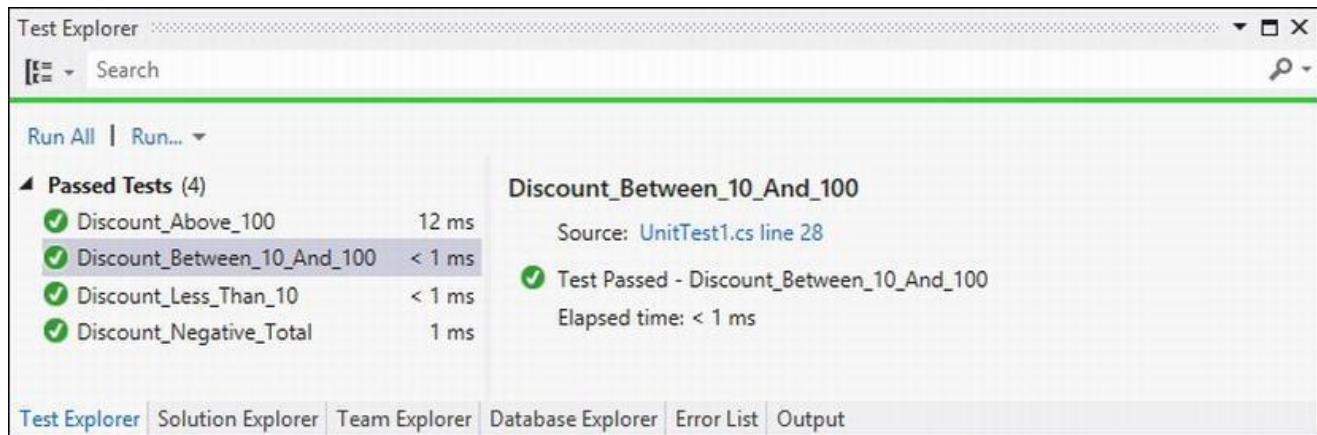
```

{
    public decimal ApplyDiscount(decimal totalParam)
    {
        if (totalParam < 0)
        {
            throw new ArgumentOutOfRangeException();
        }
        else if (totalParam > 100)
        {
            return totalParam * 0.9M;
        }
        else if (totalParam >= 10 && totalParam <= 100)
        {
            return totalParam - 5;
        }
        else
        {
            return totalParam;
        }
    }
}
}

```

Если мы нажмем на кнопку Run All в окне **Test Explorer**, результаты покажут, что мы решили проблему и что наш код прошел все тесты (см. рисунок 6-9).

**Рисунок 6-9:** Прохождение всех тестов



Мы кратко показали вам модульное тестирование, но мы также будем работать с юнит тестами в следующих главах. Поддержка модульного тестирования в Visual Studio довольно хорошо реализована, и мы рекомендуем вам изучить документацию по модульному тестированию в MSDN. Эту документацию который вы можете найти на <http://msdn.microsoft.com/enus/library/dd264975.aspx>.

## Использование Moq

Одна из причин того, что наши тесты в предыдущем разделе были настолько просты, заключается в том, что мы тестировали единственный класс, который не зависел ни от какого другого класса. Конечно, такие объекты существуют в реальных проектах, но вам также нужно будет проверять объекты, которые не могут функционировать изолированно. В таких ситуациях вы должны быть в состоянии сосредоточиться на конкретном классе или методе.

Один из хороших подходов заключается в использовании *mock-объектов*, которые симулируют функциональность реальных объектов проекта, но очень конкретным и контролируемым образом.

Mock-объекты позволяют сузить фокус тестов, так чтобы вы могли проверить только тот функционал, в котором вы заинтересованы.

Платные версии Visual Studio 2012 включают в себя поддержку создания mock-объектов благодаря функции под названием *fakes*, но мы предпочитаем использовать библиотеку Moq, которая проста, удобна и может быть использована со всеми выпусками Visual Studio, в том числе бесплатными.

## Понимание проблемы

Прежде чем начать использовать Moq мы хотим показать проблему, которые мы пытаемся исправить. В этом разделе мы собираемся провести модульное тестирование класса `LinqValueCalculator`, который мы определили в папке `Models` нашего проекта. В качестве напоминания мы представим в листинге 6-27 определение класса `LinqValueCalculator`.

### Листинг 6-27: Класс `LinqValueCalculator`

```
using System.Collections.Generic;
using System.Linq;
namespace EssentialTools.Models
{
    public class LinqValueCalculator : IValueCalculator
    {
        private IDiscountHelper discounter;
        public LinqValueCalculator(IDiscountHelper discounterParam)
        {
            discounter = discounterParam;
        }
        public decimal ValueProducts(IEnumerable<Product> products)
        {
            return discounter.ApplyDiscount(products.Sum(p => p.Price));
        }
    }
}
```

Для тестирования этого класса мы добавили новый новый юнит тест класс в тестовый проект. Вы можете сделать это, щелкнув правой кнопкой мыши по тестовому проекту в Solution Explorer. Выберите пункт Add Unit Test из всплывающего меню. Если в меню Add нет пункта Unit Test, выберите New Item и используйте шаблон Basic Unit Test. Вы можете увидеть изменения, внесенные в новый файл, который Visual Studio по умолчанию называет `UnitTest2.cs`, в листинге 6-28.

### Листинг 6-28: Добавление юнит теста для класса `ShoppingCart`

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using EssentialTools.Models;
using System.Linq;
namespace EssentialTools.Tests
{
    [TestClass]
    public class UnitTest2
    {
        private Product[] products =
            new Product {Name = "Kayak", Category = "Watersports", Price = 275M},
            new Product {Name = "Lifejacket", Category = "Watersports", Price = 48.95M},
            new Product {Name = "Soccer ball", Category = "Soccer", Price = 19.50M},
            new Product {Name = "Corner flag", Category = "Soccer", Price = 34.95M};
        [TestMethod]
        public void Sum_Products_Correctly()
        {
```

```
// arrange
var discounter = new MinimumDiscountHelper();
var target = new LinqValueCalculator(discounter);
var goalTotal = products.Sum(e => e.Price);
// act
var result = target.ValueProducts(products);
// assert
Assert.AreEqual(goalTotal, result);
}
}
```

Наша проблема состоит в том, что класс `LinqValueCalculator` зависит от реализации интерфейса `IDiscountHelper`. В этом примере мы использовали класс `minimumDiscountHelper`, который заставляет нас подумать о двух вещах.

Во-первых, наш юнит тест стал сложным и негибким. Для того чтобы создать модульный тест, который работает, мы должны принять во внимание логику скидки в реализации `IDiscountHelper`, чтобы выяснить ожидаемое значение, возвращаемое методом `ValueProducts`. А хрупкость тестов заключается в том, что наши тесты не сработают, если в реализации изменится логика скидок.

И во-вторых, что самое тревожное, мы расширили сферу нашего модульного теста таким образом, что он неявно включает в себя класс `MinimumDiscountHelper`. Если наш юнит тест не сработает, мы не будем знать, заключается ли проблема в классе `LinqValueCalculator` или же в классе `MinimumDiscountHelper`.

Модульные тесты работают лучше, когда они просты и носят целенаправленный характер, а наш текущий тест не соответствует обеим из этих характеристик. В следующих разделах мы покажем вам, как добавить и применить Moq в вашем MVC проекте, так чтобы вы смогли избежать таких проблем.

## Добавление Moq в проект Visual Studio

Так же как с Ninject ранее в этой главе, самый простой способ добавить Moq в проект MVC – это использовать интегрированную поддержку Visual Studio для NuGet, что позволяет легко установить широкий набор пакетов и без проблем их обновлять. Выберите Tools Library Package Manager Manage NuGet Packages for Solution, чтобы открыть диалоговое окно NuGet пакетов.

Выберите **Online** в левой панели и введите **Moq** в поле поиска в правом верхнем углу диалогового окна. Вы увидите ряд пакетов **Moq**, похожих на те, что показаны на рисунке 6-10.

**Рисунок 6-10:** Выбор Moq из пакетов NuGet



Нажмите кнопку `Install` для библиотеки Moq, и Visual Studio загрузит библиотеку и установит ее в ваш проект. Вы увидите Moq в разделе `References` проекта.

#### **Внимание**

*Мы будем использовать Moq в проекте модульного тестирования, а не в MVC проекте, поэтому убедитесь, что вы добавить библиотеку в правильный проект.*

Опять же, мы рекомендуем вам использовать NuGet, чтобы установить Moq, но мы загрузили библиотеку непосредственно с веб-сайта проекта (<http://code.google.com/p/moq>) и установили ее вручную, чтобы размер исходного кода, прилагаемого к этой книге, был минимизирован по максимуму. Мы установили ее вручную, выбрав `Add Reference` из Visual Studio меню `Project`, нажали на кнопку `Browse`, потом перешли к архиву, извлекли содержимое и выбрали файл `moq.dll`.

#### **Примечание**

*Мы хотим повторить то, что мы вам сказали, когда вручную устанавливали Ninject: единственная причина, почему мы добавили библиотеку Moq вручную, заключается в том, чтобы сохранить размер исходного кода, прилагаемого к этой книге. В реальных проектах мы используем NuGet, когда несколько дополнительных мегабайт данных не имеют никакого значения, и мы также рекомендуем вам использовать NuGet.*

## **Добавление mock-объекта в юнит тест**

Добавление mock-объекта в модульный тест обозначает, что вы говорите Moq, с каким объектом вы хотите работать, настраивая его поведение, а затем применяя объект к тестируемой цели. В листинге 6-29 показано, как мы добавили mock-объект в наш юнит тест для `LinqValueCalculator`.

#### **Листинг 6-29:** Использование mock-объекта в юнит teste

```
using EssentialTools.Models;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;
using System.Linq;
namespace EssentialTools.Tests
{
    [TestClass]
    public class UnitTest2
    {
        private Product[] products = {
            new Product {Name = "Kayak", Category = "Watersports", Price = 275M},
            new Product {Name = "Lifejacket", Category = "Watersports", Price = 48.95M},
            new Product {Name = "Soccer ball", Category = "Soccer", Price = 19.50M},
            new Product {Name = "Corner flag", Category = "Soccer", Price = 34.95M}
        };
        [TestMethod]
        public void Sum_Products_Correctly()
        {
            // arrange
            Mock<IDiscountHelper> mock = new Mock<IDiscountHelper>();
            mock.Setup(m => m.ApplyDiscount(It.IsAny<decimal>()))
                .Returns<decimal>(total => total);
            var target = new LinqValueCalculator(mock.Object);
            // act
            var result = target.ValueProducts(products);
```

```

    // assert
    Assert.AreEqual(products.Sum(e => e.Price), result);
}
}
}

```

Синтаксис использования Moq немного странный, если вы впервые видите его, так что мы пройдем по каждой стадии процесса.

#### *Совет*

*Имейте в виду, что существует целый ряд различных mock-библиотек, так что есть шанс, что вы можете найти альтернативу, если вам не нравится то, как работает Moq: хотя на самом деле Moq – легкая библиотека для использования. Есть другие популярные библиотеки, документация по которым составляет сотни страниц.*

---

## Создание mock-объекта

Первым делом надо сообщить Moq, с каким mock-объектом вы хотите работать. Moq в значительной степени зависит от параметров универсального типа, и вы видите это в том, как мы говорим Moq, что мы хотим создать mock-реализацию `IDiscountHelper`:

```
Mock<IDiscountHelper> mock = new Mock<IDiscountHelper>();
```

Мы создаем строго типизированный объект `Mock<IDiscountHelper>`, который говорит библиотеке Moq, какой тип он будет обрабатывать: конечно, это интерфейс `IDiscountHelper` для наших модульных тестов, но это может быть любой тип, который вы хотите изолировать для улучшения модульных тестов.

## Выбор метода

В дополнение к созданию строго типизированного `Mock` объекта мы также должны указать, каким способом он ведет себя: это сердце mock-процесса и это позволяет вам убедиться в том, что вы установили базовое поведение в mock-объект, который вы сможете использовать для тестирования функциональности вашего целевого объекта. Это выражение юнит теста, которое устанавливает желаемое поведение:

```
mock.Setup(m => m.ApplyDiscount(It.IsAny<decimal>())).Returns<decimal>(total => total);
```

Мы используем метод `Setup`, чтобы добавить метод в наш mock-объект. Moq работает с использованием LINQ и лямбда-выражений. Когда мы вызываем метод `Setup`, Moq передает нам интерфейс, который мы попросили реализовать. В этом есть некая магия LINQ, в которую мы не собираемся углубляться, но она позволяет нам выбрать метод, который мы хотим определить, через лямбда-выражения. Для нашего модульного теста мы хотим определить поведение метода `ApplyDiscount`, который является единственным методом в интерфейсе `IDiscountHelper` и методом, которым нам нужно протестировать класс `LinqValueCalculator`.

Мы также должны сказать Moq, в каких значениях параметров мы заинтересованы, что мы и делаем, используя класс `It`, который мы выделили:

```
mock.Setup(m => m.ApplyDiscount(It.IsAny<decimal>())).Returns<decimal>(total => total);
```

Класс `It` определяет ряд методов, которые используются с параметрами универсального типа. В данном случае мы назвали метод `IsAny`, используя `decimal` как универсальный тип. Это говорит Moq, что поведение, которое мы определяем, следует применять всякий раз, когда вызывается `ApplyDiscount` с любым десятичным значением. В таблице 6-3 представлены методы класса `It`, и все они являются статическими.

**Таблица 6-3:** Методы класса `It`

Метод	Описание
<code>Is&lt;T&gt;(predicate)</code>	Задает значения типа <code>T</code> , которое заставляет предикат вернуть <code>true</code> (см. листинг 6-30 для примера).
<code>IsAny&lt;T&gt;()</code>	Задает любое значение типа <code>T</code> .
<code>IsInRange&lt;T&gt;(min, max, kind)</code>	Срабатывает, если параметр находится между определенными значениями и относится к типу <code>T</code> . Последний параметр является значением перечисления <code>Range</code> и может быть <code>Inclusive</code> или <code>Exclusive</code> .
<code>IsRegex(expr)</code>	Соответствует строковому параметру, если он соответствует указанному регулярному выражению.

Мы покажем вам более сложный пример далее в этой главе, где используется другие `It` методы, но на данный момент мы рассмотрим метод `IsAny<decimal>`, который позволяет нам работать с любым десятичным значением.

### Определение результата

Метод `Returns` позволяет определить результат, который будет возвращен, когда будет вызван наш mock-метод. Мы указали тип результата с помощью параметра типа и указали сам результат с помощью лямбда-выражения. Вот как мы это сделали:

```
mock.Setup(m => m.ApplyDiscount(It.IsAny<decimal>())).Returns<decimal>(total => total);
```

Вызывая метод `Returns` с параметром `decimal` (т.е. `Returns<decimal>`), мы говорим Moq, что собирается вернуть значение `decimal`. Для лямбда-выражения Moq передает нам значение типа, которое мы получили в методе `ApplyDiscount`: в этом примере мы создали *переходный* метод, в который мы возвращаем значение, переданное mock-методу `ApplyDiscount` без выполнения любых операций с ним. Это самый простой вид mock-метода, но в ближайшее время мы покажем вам более сложные примеры.

### Использование mock-объекта

Последний шаг заключается в использовании mock-объекта в юнит teste, что мы и делаем, считывая свойство `Object` объекта `Mock<IDiscountHelper>`:

```
var target = new LinqValueCalculator(mock.Object);
```

Итак, в нашем примере свойство `Object` возвращает реализацию интерфейса `IDiscountHelper`, где метод `ApplyDiscount` возвращает значение параметра `decimal`, который ему передается.

Наши тесты очень легко выполнить, потому что мы сами можем вывести цены объектов `Product` и убедиться, что мы получаем то же самое значение от объекта `LinqValueCalculator`:

```
Assert.AreEqual(products.Sum(e => e.Price), result);
```

Преимущество использования Moq таким образом заключается в том, что наш юнит тест проверяет только поведение объекта `LinqValueCalculator` и не зависит от какой-либо реальной реализации интерфейса `IDiscountHelper` в папке `Models`. Это означает, что если наши тесты не сработают, мы будем знать, что проблема заключается либо в реализации `LinqValueCalculator` или в том, как мы создали mock-объект. И решение проблемы в этих двух направления гораздо проще, нежели работа с цепочкой реальных объектов и взаимодействиями между ними.

## Создание более сложного mock-объекта

Мы показали вам очень простой mock-объект в предыдущем разделе, но часть всей прелести Moq заключается в возможности быстро создавать сложные виды поведения для тестирования различных ситуаций. В листинге 6-30 мы добавили новый юнит тест в файл `UnitTest2.cs`, который представляет более сложную реализацию интерфейса `IDiscountHelper`: на самом деле, мы использовали Moq для моделирования поведения класса `MinimumDiscountHelper`.

### Листинг 6-30: Использование Moq для класса `MinimumDiscountHelper`

```
using EssentialTools.Models;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;
using System.Linq;
namespace EssentialTools.Tests
{
    [TestClass]
    public class UnitTest2
    {
        private Product[] products =
            new Product {Name = "Kayak", Category = "Watersports", Price = 275M},
            new Product {Name = "Lifejacket", Category = "Watersports", Price = 48.95M},
            new Product {Name = "Soccer ball", Category = "Soccer", Price = 19.50M},
            new Product {Name = "Corner flag", Category = "Soccer", Price = 34.95M};
        ;
        [TestMethod]
        public void Sum_Products_Correctly()
        {
            // arrange
            Mock<IDiscountHelper> mock = new Mock<IDiscountHelper>();
            mock.Setup(m => m.ApplyDiscount(It.IsAny<decimal>()))
                .Returns<decimal>(total => total);
            var target = new LinqValueCalculator(mock.Object);
            // act
            var result = target.ValueProducts(products);
            // assert
            Assert.AreEqual(products.Sum(e => e.Price), result);
        }
        private Product[] createProduct(decimal value)
        {
            return new[] { new Product { Price = value } };
        }
        [TestMethod]
        [ExpectedException(typeof(System.ArgumentOutOfRangeException))]
        public void Pass_Through_Variable_Discounts()
        {
            // arrange
            Mock<IDiscountHelper> mock = new Mock<IDiscountHelper>();
            mock.Setup(m => m.ApplyDiscount(It.IsAny<decimal>()))
                .Returns<decimal>(total => total);
            mock.Setup(m => m.ApplyDiscount(It.Is<decimal>(v => v == 0)))
                .Throws<System.ArgumentOutOfRangeException>();
            mock.Setup(m => m.ApplyDiscount(It.Is<decimal>(v => v > 100)))
                .Returns<decimal>(total => (total * 0.9M));
            mock.Setup(m => m.ApplyDiscount(It.IsInRange<decimal>(10, 100, Range.Inclusive)))

```

```
    .Returns<decimal>(total => total - 5);
var target = new LinqValueCalculator(mock.Object);
// act
decimal FiveDollarDiscount = target.ValueProducts(createProduct(5));
decimal TenDollarDiscount = target.ValueProducts(createProduct(10));
decimal FiftyDollarDiscount = target.ValueProducts(createProduct(50));
decimal HundredDollarDiscount = target.ValueProducts(createProduct(100));
decimal FiveHundredDollarDiscount = target.ValueProducts(createProduct(500));
// assert
Assert.AreEqual(5, FiveDollarDiscount, "$5 Fail");
Assert.AreEqual(5, TenDollarDiscount, "$10 Fail");
Assert.AreEqual(45, FiftyDollarDiscount, "$50 Fail");
Assert.AreEqual(95, HundredDollarDiscount, "$100 Fail");
Assert.AreEqual(450, FiveHundredDollarDiscount, "$500 Fail");
target.ValueProducts(createProduct(0));
}
}
```

С точки модульного тестирования тиражирование ожидаемое поведение одного из других классов модели вроде бы странная вещь, но это прекрасная демонстрация того, как мы можем использовать Moq.

Вы видите, что мы определили четыре различных вида поведения для метода `ApplyDiscount`, основываясь на значении параметра, которые мы получаем. Простейшим из них является тот, который возвращает значение любого `decimal` значения, например:

```
mock.Setup(m => m.ApplyDiscount(It.IsAny<decimal>())).Returns<decimal>(total => total);
```

Это то же самое поведение, которое было использовано в предыдущем примере, и мы включили его сюда, потому что порядок, в котором вы вызываете метод `Setup`, влияет на поведение mock-объекта. Moq оценивает данные ему виды поведения в обратном порядке, так что самые последние вызовы метода `Setup` считаются первыми. Это обозначает, что вы должны создавать mock-поведения в порядке от более общих к более конкретным. Условие `It.IsAny<decimal>` является наиболее общим условием, которое мы определили в данном примере, и поэтому мы применяем его первым. Если мы изменим порядок вызовов `Setup`, такое поведение может охватить все вызовы метода `ApplyDiscount` и сгенерировать неправильный результат.

#### **Использование токс-объектов для указанных значений (и получение исключения)**

Для нашего второго вызова метода `Setup` мы использовали метод `It.IsAny<T>()`:

```
    .Throws<System.ArgumentOutOfRangeException>();
```

Предикат, который мы передали методу `Is`, возвращает `true`, если значение, переданное методу `ApplyDiscount` равно 0. Вместо того чтобы вернуть результат, мы использовали метод `Throws`, который заставляет Moq выбросить новый экземпляр исключения, которое мы указываем с параметром типа.

Мы также используем метод `Is`, чтобы охватить значения больше, чем 100:

```
mock.Setup(m => m.ApplyDiscount(It.Is<decimal>(v => v > 100)))
    .Returns<decimal>(total => (total * 0.9M));
```

Использование метода `it.Is` – это наиболее гибкий способ создания определенного поведения для различных значений параметра, потому что вы можете использовать любой предикат, который

возвращает `true` или `false`. Этот метод мы используем чаще всего при создании сложных mock-объектов.

## Использование mock-объектов для диапазона значений

Наше последнее использование объекта `It` связано с методом `IsInRange`, который позволяет нам охватить определенный диапазон значений параметров:

```
mock.Setup(m => m.ApplyDiscount(It.IsInRange<decimal>(10, 100, Range.Inclusive)))
    .Returns<decimal>(total => total - 5);
```

Мы включили это для полноты картины, но в наших собственных проектах, мы, как правило, используем метод `Is` и предикат, который делает вот это:

```
mock.Setup(m => m.ApplyDiscount(It.Is<decimal>(v => v >= 10 && v <= 100)))
    .Returns<decimal>(total => total - 5);
```

Результат тот же, но мы считаем, что работа через предикат отличается более гибким подходом. Moq обладает целым рядом чрезвычайно полезных функций, и вы можете ознакомиться с ними на <http://code.google.com/p/moq/wiki/QuickStart>.

## Резюме

В этой главе мы рассмотрели три инструмента, которые являются важными для эффективной MVC разработки: Ninject, встроенная поддержка Visual Studio для модульного тестирования и Moq. Есть много альтернатив, как с открытым кодом, так и коммерческих, для всех этих трех инструментов, и вы не будете обделены, если вам не понравятся те инструменты, которые используем мы.

Возможно, вам не нравится TDD или модульное тестирование или вам нравится использование DI и создание mock-объектов вручную. Это полностью ваш выбор. Мы считаем, что есть некоторые существенные выгоды при использовании всех трех инструментов в цикле разработки. Если вы не решаетесь принять их, потому что вы никогда их не пробовали, мы просим вас не закрывать глаза, а последовать за нами, хотя бы до конца этой книги.

# SportsStore: реальное приложение

В предыдущих главах мы создали быстрое и простое приложение MVC. Мы рассмотрели паттерн MVC, вспомнили наиболее существенные функции C# и описали различные инструменты, которые пригодятся разработчикам MVC. Теперь пришло время объединить все пройденное и создать простое и в то же время реалистичное приложение для электронной коммерции.

Наше приложение – SportsStore – мы создадим по классической схеме, которая применяется во всех интернет-магазинах. Мы создадим онлайн-каталог товаров, который можно просматривать по категориям и страницам, корзину, где можно добавлять и удалять товары, и кассу, где можно ввести информацию о доставке. Мы также создадим область администрирования, в которой будут возможности для управления каталогом – создать, прочитать, обновить и удалить (create, read, update и delete - CRUD), для которой установим защиту, чтобы только залогинившиеся администраторы могли вносить изменения.

Данное приложение мы создаем не просто для демонстрации. Напротив, мы создаем реалистичное и работоспособное приложение, используя продвинутые техники. Когда мы будем строить необходимые для него уровни инфраструктуры, процесс может показаться вам немного медленным. Несомненно, получить базовую функциональность можно гораздо быстрее с помощью Web Forms, просто перетаскивая элементы управления, связанные непосредственно с базой данных. Но начальное вложение в приложение MVC приносит неплохие дивиденды, так как в итоге мы получим поддерживаемый, расширяемый, хорошо структурированный код с возможностями для модульного тестирования. Мы сможем ускориться, когда базовая инфраструктура будет готова.

## Модульное тестирование

Мы много говорили о легкости модульного тестирования в MVC, и о нашей убежденности в том, что модульное тестирование – это важная часть процесса разработки. Данный момент вы сможете проследить на протяжении всей этой части, потому что мы включили в нее подробную информацию о тестах и техниках, которые относятся к основным функциям MVC.

Но мы знаем, что не все разделяют это убеждение. Если вы не хотите проводить модульное тестирование – мы не против. Таким образом, всю информацию о модульном тестировании или TDD мы будем выносить в специальные блоки, такие как этот. Если вас не интересует модульное тестирование, вы можете пропустить эти секции, и приложение SportsStore от этого не пострадает. Вы сможете воспользоваться всеми техническими преимуществами ASP.NET MVC и без модульного тестирования, но, тем не менее, оно поможет вам воспользоваться преимуществами MVC и в разработке, и в тестировании.

Некоторым возможностям MVC, которые мы собираемся использовать, посвящены отдельные главы далее в этой книге. Чтобы не дублировать здесь все, мы изложим достаточно информации, чтобы вы смогли понять суть примера, и укажем, в какой главе искать подробную информацию.

Мы отдельно разберем каждый шаг, необходимый для создания приложения, чтобы вы могли видеть, как возможности MVC работают вместе. Будьте особенно внимательны, когда мы будем создавать представления. Вы получите несколько странные результаты, если будете использовать не те опции, которые используем мы. Чтобы помочь вам, мы включали картинки с диалоговым окном Add View каждый раз, когда добавляли представление в проект.

# Начинаем

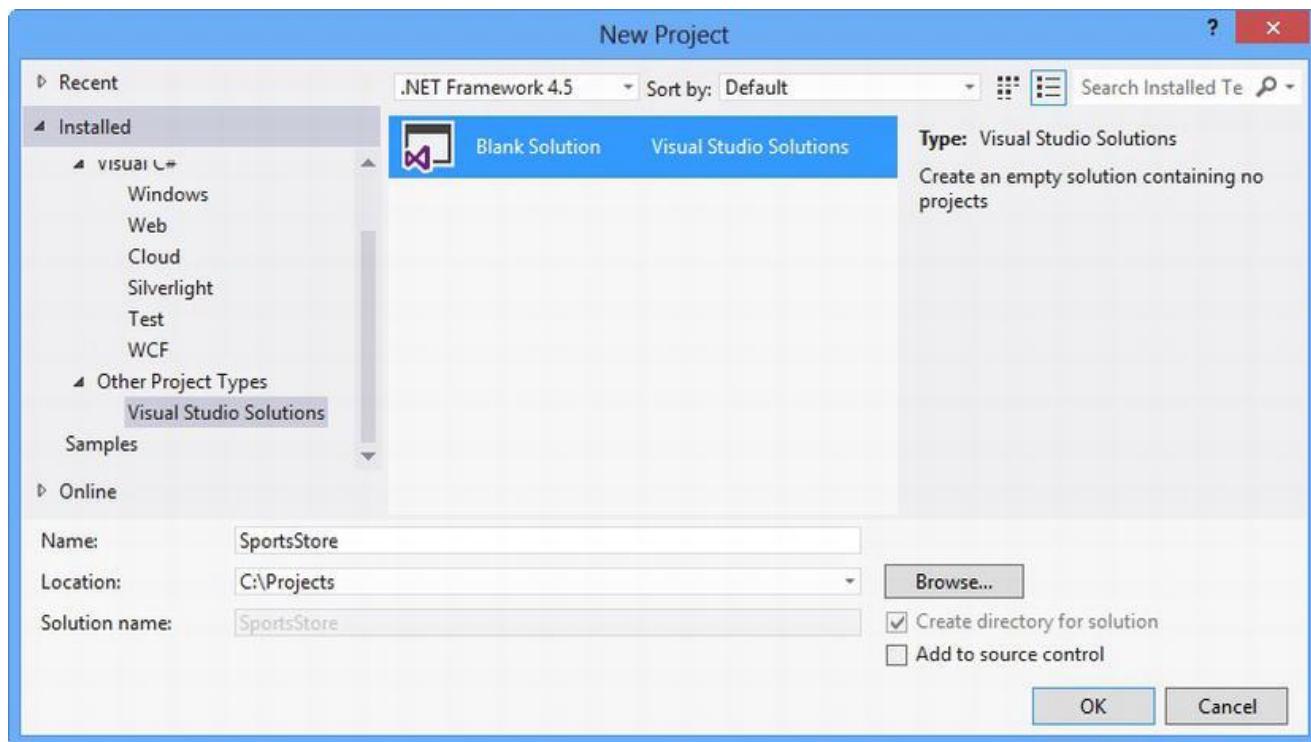
Если вы планируете кодировать SportsStore на своем компьютере, вам нужно будет установить программное обеспечение, описанное в главе 2. Вы также можете взять SportsStore из архива с кодом, который прилагается к этой книге (доступен на [apress.com](http://apress.com)). Мы включали скриншоты проекта после того, как добавляли основные функции, чтобы можно было проследить, как изменялось приложение в процессе нашей работы над ним.

Создавать приложение вместе с нами вам, конечно, не обязательно. Мы постарались сделать скриншоты и листинги с кодом настолько доступными, насколько это возможно, на случай, если вы читаете эту книгу в поезде, кафе или тому подобное.

## Создаем решение и проекты Visual Studio

Мы собираемся создать решение Visual Studio, которое включает три проекта. Один проект будет содержать нашу доменную модель, второй будет нашим MVC-приложением, а третий будет содержать модульные тесты. Для начала мы создадим новое решение Visual Studio под названием SportsStore, используя шаблон Blank Solution, который можно найти в разделе Other Project Types диалога New Project, как показано на рисунке 7-1. Для создания решения нажмите OK.

Рисунок 7-1: Создаем новое решение Visual Studio



Решение Visual Studio является контейнером для одного или нескольких проектов. Для нашего примера нам понадобится три проекта, которые мы описали в таблице 7-1. Чтобы добавить проект, щелкните правой кнопкой по вкладке Solution в Solution Explorer и выберите Add New Project в контекстном меню.

**Таблица 7-1:** Три проекта SportsStore

Название проекта	Шаблон проекта	Цель проекта
SportsStore.Domain	Class Library	Содержит доменные объекты и логику; поддерживает <b>механизм хранения с помощью хранилища</b> , созданного используя Entity Framework.
SportsStore.WebUI	ASP.NET MVC 4 Web Application (Выберите Basic, когда будет предложено выбрать шаблон проекта)	Содержит контроллеры и представления; выступает в качестве интерфейса приложения.
SportsStore.UnitTests	Unit Test Project	Содержит модульные тесты для двух других проектов.

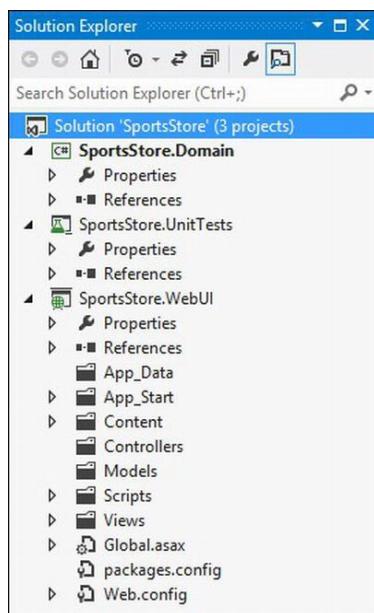
Мы используем версию `Basic` шаблона ASP.NET MVC 4 Web Application, поскольку она содержит несколько полезных дополнений: набор часто используемых библиотек JavaScript и папку `Scripts` (в том числе JQuery, JQuery UI, Knockout и Modernizr), базовый макет в папке `Views` и несколько CSS стилей в папке `Content` (в том числе стили валидации формы, которые мы добавляли вручную в главе 2).

### Примечание

Файлы библиотеки JavaScript, которые включены в шаблон `Basic`, описаны в главе 24. Стив является создателем библиотеки Knockout. Адам описал Knockout в своей книге *Pro JavaScript for Web Apps* (Apress, 2012), наряду с Modernizr и некоторыми другими важными библиотеками JavaScript и интерфейсами HTML5. Адам также писал о JQuery, JQuery UI и JQuery Mobile в своей книге [Pro JQuery](#) (Apress, 2012). Как вы, возможно, догадываетесь, мы оба испытываем глубокий интерес к этой области и считаем, что невозможно создать хорошее серверное приложение, не уделяя должного внимания клиентской стороне.

Вы можете удалить файл `Class1.cs` в проекте `SportsStore.Domain` - мы не будем его использовать. Когда вы это сделаете, окно Solution Explorer должно выглядеть так, как показано на рисунке 7-2.

**Рисунок 7-2:** Проекты отображаются в окне Solution Explorer



Чтобы сделать отладку легче, щелкните правой кнопкой мыши по проекту `SportsStore.WebUI` и выберите команду `Set as Startup Project` в контекстном меню (имя выделится полужирным шрифтом). Это означает, что, когда вы выберите `Start Debugging` или `Start without Debugging` в меню `Debug`, будет запущен именно этот проект.

## Добавляем ссылки

Нам нужно добавить ссылки на библиотеки инструментов, которые мы собираемся использовать для двух проектов. Самый простой способ добавить библиотеки - это кликнуть правой кнопкой мыши по каждому проекту, выбрать пункт `Manage NuGet Packages`, который вызывает диалог NuGet, и найти и установить необходимые библиотеки.

Нам также нужно установить зависимости между проектами. Кликните правой кнопкой мыши по каждому проекту в окне `Solution Explorer`, выберите пункт `Add Reference` и добавьте ссылки на необходимые библиотеки инструментов или другие проекты из раздела `Solution`.

Вы можете просмотреть подробную информацию о проектах, пакетах NuGet и зависимостях от других проектов в таблице 7-2.

### Внимание

Не жалейте времени на то, чтобы установить эти отношения должным образом. Если вы установите неправильные библиотеки и ссылки на проекты, у вас будут проблемы в дальнейшей работе над проектом.

**Таблица 7-2:** Необходимые зависимости проекта

Название проекта	Зависимости инструментов	Зависимости проектов	Ссылки Microsoft
<code>SportsStore.Domain</code>	None	None	<code>System.Web.Mvc</code> <code>System.ComponentModel.DataAnnotations</code>
<code>SportsStore.WebUI</code>	<code>Ninject</code> , <code>Moq</code>	<code>SportsStore.Domain</code>	None
<code>SportsStore.UnitTests</code>	<code>Ninject</code> , <code>Moq</code>	<code>SportsStore.Domain</code> , <code>SportsStore.WebUI</code>	<code>System.Web.Mvc</code> , <code>System.Web</code> , <code>Microsoft.CSharp</code>

Чтобы добавить ссылку на сборку `System.Web.Mvc` в проект `SportsStore.UnitTests`, кликните правой кнопкой мыши по имени проекта в `Solution Explorer`, выберите пункт `Add Reference` и перейдите в раздел `Assemblies - Extensions`. Вы найдете несколько ссылок на сборки под названием `System.Web.Mvc`. Убедитесь, что вы добавляете версию 4.0.0. Если вы выберите более раннюю версию, у вас могут возникнуть проблемы при тестировании функций, которые были изменены в последней версии. Другие ссылки Microsoft добавляются таким же образом, но находятся в разделе `Assemblies - Framework`.

## Устанавливаем контейнер DI

В главе 6 мы разобрали, как с помощью `Ninject` создать пользовательский DR (dependency resolver), который используется платформой MVC для создания экземпляров объектов в приложении. В этом примере мы собираемся поступить иначе, а именно - создать *пользовательскую фабрику контроллеров*. Это проиллюстрирует одну из многих точек расширения MVC Framework, где вы можете добавить пользовательский код, чтобы изменить поведение платформы или, как мы делаем здесь, ограничить DI и использовать его только в одной части приложения. Обычно DR используется для работы с DI и пользовательской фабрикой контроллеров, чтобы изменить то, как размещаются

классы контроллера (к этой теме мы еще вернемся в главе 15). В этом примере приложения мы собираемся использовать только фабрику контроллеров, так как хотим продемонстрировать вам как можно больше различных аспектов платформы MVC.

Создайте в проекте SportsStore.WebUI новую папку под названием Infrastructure, а затем создайте класс NinjectControllerFactory и отредактируйте файл класса так, чтобы он соответствовал листингу 7-1.

#### Листинг 7-1: Класс NinjectControllerFactory

```
using System;
using System.Web.Mvc;
using System.Web.Routing;
using Ninject;

namespace SportsStore.WebUI.Infrastructure
{
    // реализация пользовательской фабрики контроллеров,
    // наследуясь от фабрики используемой по умолчанию
    public class NinjectControllerFactory : DefaultControllerFactory
    {
        private IKernel ninjectKernel;
        public NinjectControllerFactory()
        {
            // создание контейнера
            ninjectKernel = new StandardKernel();
            AddBindings();
        }

        protected override IController GetControllerInstance(RequestContext requestContext,
Type controllerType)
        {
            // получение объекта контроллера из контейнера
            // используя его тип
            return controllerType == null
                ? null
                : (IController)ninjectKernel.Get(controllerType);
        }

        private void AddBindings()
        {
            // конфигурирование контейнера
        }
    }
}
```

Мы еще не добавляли привязки Ninject, но когда они нам понадобятся, мы можем сделать это с помощью метода AddBindings. Необходимо сообщить MVC, что для создания объектов контроллера мы хотим использовать класс NinjectController. Для этого мы делаем дополнение в методе Application\_Start в файле Global.asax.cs проекта SportsStore.WebUI, которое выделено жирным шрифтом в листинге 7-2.

#### Листинг 7-2: Регистрируем NinjectControllerFactory в MVC Framework

```
using SportsStore.WebUI.Infrastructure;
using System.Web.Http;
using System.Web.Mvc;
using System.Web.Optimization;
using System.Web.Routing;

namespace SportsStore.WebUI
{
    public class MvcApplication : System.Web.HttpApplication
```

```

{
    protected void Application_Start()
    {
        AreaRegistration.RegisterAllAreas();

        WebApiConfig.Register(GlobalConfiguration.Configuration);
        FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
        RouteConfig.RegisterRoutes(RouteTable.Routes);
        BundleConfig.RegisterBundles(BundleTable.Bundles);

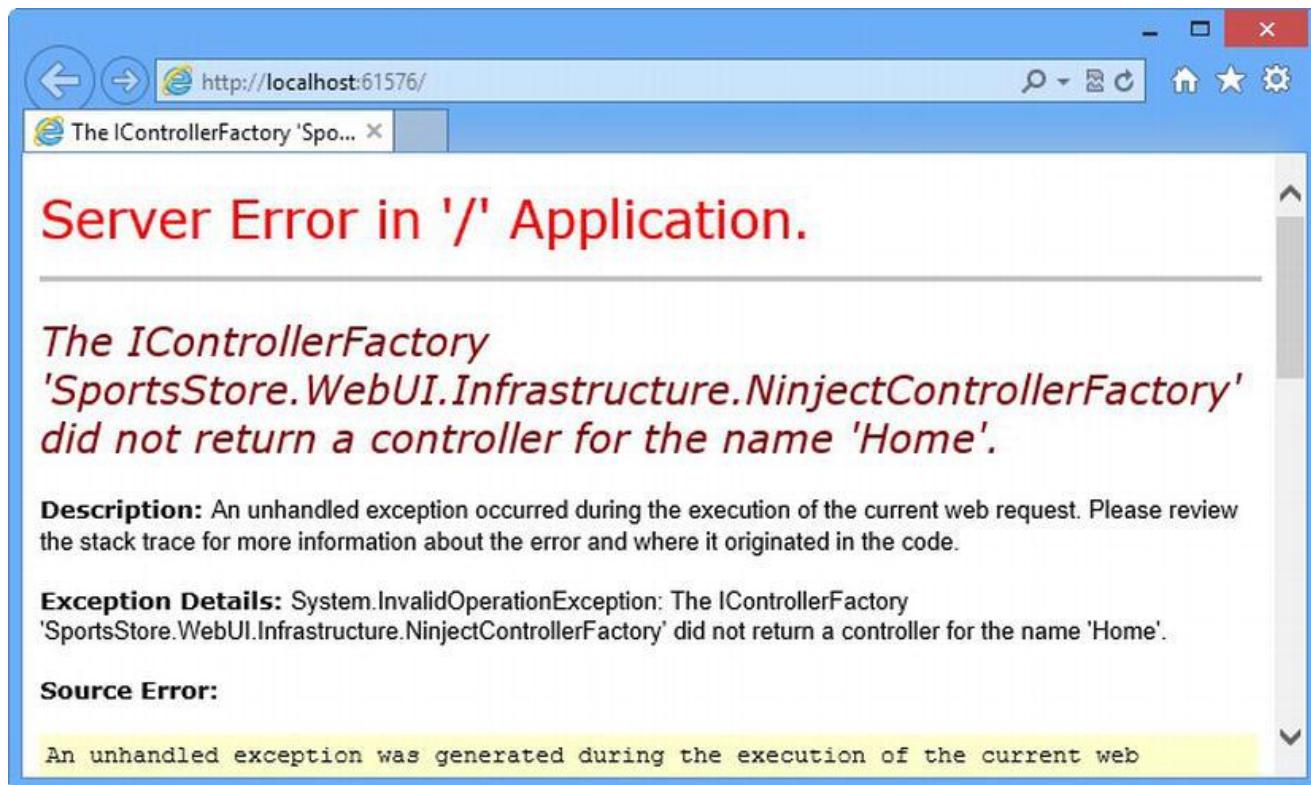
        ControllerBuilder.Current.SetControllerFactory(new NinjectControllerFactory());
    }
}
}

```

## Запускаем приложение

Если вы выберите команду Start Debugging в меню Debug, то получите страницу с сообщением об ошибке. Это произойдет потому, что вы запросили URL, связанный с контроллером, для которого у Ninject нет привязки, что показано на рисунке 7-3.

**Рисунок 7-3:** Страница с сообщением об ошибке

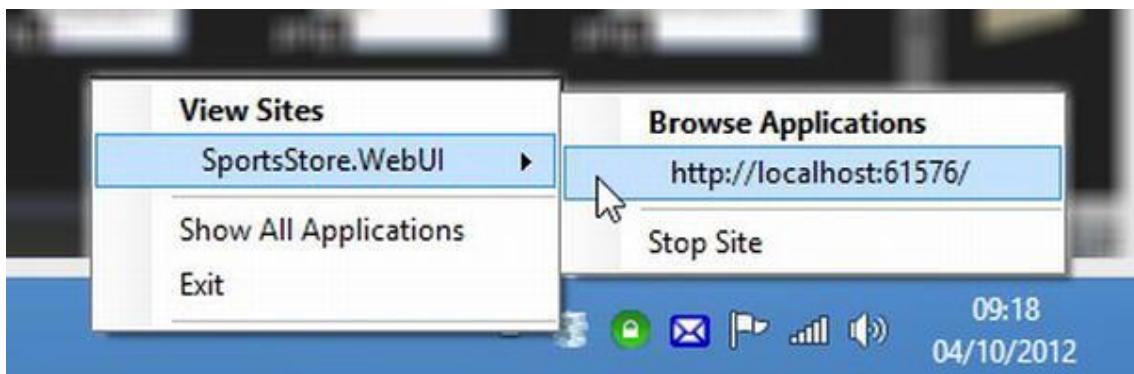


Если вы видите это окно, то Visual Studio 2012 и ASP.NET MVC у вас настроены и работают – правильно. Вы можете остановить отладку, закрыв окно браузера, если ваш браузер по умолчанию – Internet Explorer. Если нет, то переключитесь в Visual Studio и выберите пункт Stop Debugging в меню Debug.

## Упрощенная отладка

Когда вы запускаете проект из меню Debug, Visual Studio откроет новое окно браузера для отображения приложения. Чтобы ускорить процесс, вы можете открыть приложение в отдельном окне. Если вы уже запускали отладчик хотя бы один раз, кликните правой кнопкой мыши ярлык IIS

Express в системном трее и выберите URL для вашего приложения в контекстном меню, как показано на следующем скриншоте.



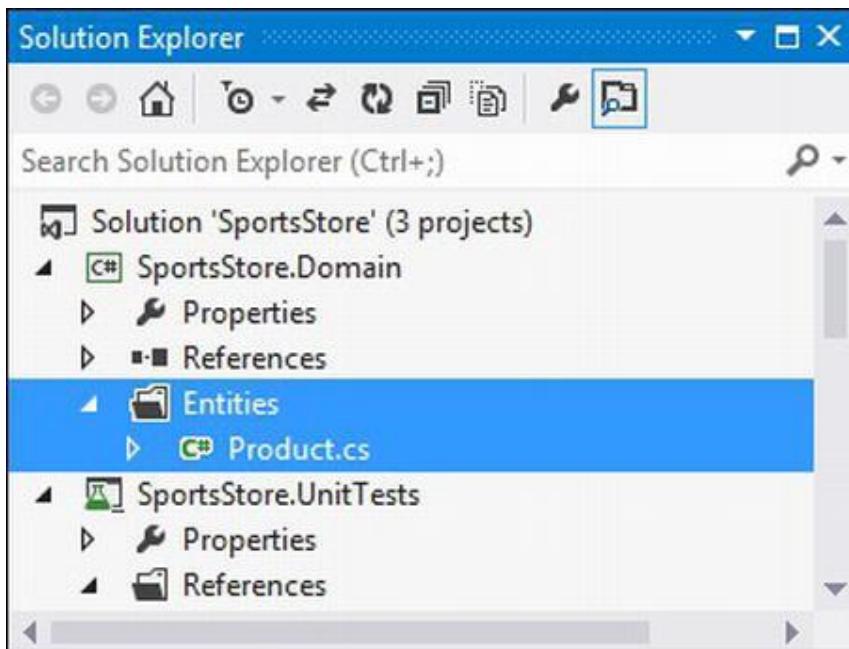
Таким образом, вам не нужно будет запускать новую сессию отладки после каждого изменения, чтобы увидеть эффект. Вы просто компилируете решение в Visual Studio, нажав F6 или выбрав Build - Build Solution, а затем перезагружаете страницу в браузере.

## Создаем доменную модель

Мы начнем с создания доменной модели. Практически все в приложении MVC вращается вокруг доменной модели, так что она и будет нашей стартовой площадкой.

Поскольку мы разрабатываем приложение для электронной коммерции, очевидно, самой главной доменной сущностью у нас будет товар. Создайте новую папку под названием `Entities` в проекте `SportsStore.Domain`, а в ней - новый класс C# под названием `Product`. Искомая структура показана на рисунке 7-4.

Рисунок 7-4: Создаем класс Product



Содержание класса `Product` вам уже известно, так как мы собираемся использовать тот же класс, что и в предыдущих главах. Отредактируйте файл класса `Product`, чтобы он соответствовал листингу 7-3.

### Листинг 7-3: Файл класса `Product`

```
namespace SportsStore.Domain.Entities
{
    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 { get; set; }
    }
}
```

Мы следуем соглашению, по которому мы определяем доменную модель в отдельном проекте Visual Studio, а это означает, что класс должен быть помечен как `public`. Следовать этому соглашению не обязательно, но мы считаем, что оно помогает сохранять изоляцию модели от контроллеров.

## Создаем абстрактное хранилище

Разумеется, нам нужно каким-то образом получать объекты `Product` из базы данных. Как мы уже объяснили в главе 3, мы хотим держать логику хранения изолированно от объектов доменной модели, и для этого мы будем использовать шаблон хранилища. Сейчас мы не будем думать о том, как мы собираемся реализовать хранение, и начнем с того, что определим для него интерфейс.

Создайте новую папку верхнего уровня в проекте `SportsStore.Domain` под названием `Abstract` и новый интерфейс под названием `IProductsRepository`, содержание которого показано в листинге 7-4. Чтобы добавить новый интерфейс, кликните правой кнопкой мыши папку `Abstract`, выберите `Add - New Item` и шаблон `Interface`.

### Листинг 7-4: Файл интерфейса `IProductRepository`

```
using System.Linq;
using SportsStore.Domain.Entities;

namespace SportsStore.Domain.Abstract
{
    public interface IProductRepository
    {
        IQueryable<Product> Products { get; }
    }
}
```

Здесь используется интерфейс `IQueryable<T>`, который позволяет получить последовательность объектов `Product` и не требует указаний на то, как и где хранятся данные или как следует их извлекать. Класс, который использует интерфейс `IProductRepository`, может получить объекты `Product`, не зная того, где они содержатся или каким образом будут ему поставлены. Это и есть суть шаблона хранилища. Мы будем возвращаться к этому интерфейсу на протяжении всего процесса разработки, чтобы добавлять новые функции.

## Создаем имитированное хранилище

Определив абстрактный интерфейс, мы можем реализовать механизм хранения и подключить его к базе данных. Мы сделаем это позже в этой главе. Чтобы начать писать другие части приложения, мы

собираемся создать имитированную реализацию интерфейса `IProductRepository`. Это мы сделаем в методе `AddBindings` класса `NinjectControllerFactory` в проекте `SportsStore.WebUI`, как показано в листинге 7-5.

#### Листинг 7-5: Добавляем имитированную реализацию `IProductRepository`

```
using System;
using System.Web.Mvc;
using System.Web.Routing;
using Ninject;
using SportsStore.Domain.Entities;
using SportsStore.Domain.Abstract;
using System.Collections.Generic;
using System.Linq;
using Moq;

namespace SportsStore.WebUI.Infrastructure
{
    public class NinjectControllerFactory : DefaultControllerFactory
    {
        private IKernel ninjectKernel;
        public NinjectControllerFactory()
        {
            ninjectKernel = new StandardKernel();
            AddBindings();
        }

        protected override IController GetControllerInstance(RequestContext requestContext,
Type controllerType)
        {
            return controllerType == null
                ? null
                : (IController)ninjectKernel.Get(controllerType);
        }
        private void AddBindings()
        {
            Mock<IProductRepository> mock = new Mock<IProductRepository>();
            mock.Setup(m => m.Products).Returns(new List<Product> {
                new Product { Name = "Football", Price = 25 },
                new Product { Name = "Surf board", Price = 179 },
                new Product { Name = "Running shoes", Price = 95 }
            }.AsQueryable());

            ninjectKernel.Bind<IProductRepository>().ToConstant(mock.Object);
        }
    }
}
```

Для этого дополнения мы должны были добавить в файл несколько пространств имен, но процесс, который создает имитированное хранилище, использует те же самые техники `Moq`, которые мы рассмотрели в главе 4. `AsQueryable` является методом расширения LINQ, который преобразует `IEnumerable<T>` в `IQueryable<T>`. Это необходимо для соответствия сигнатуры интерфейса.

Мы используем метод `ToConstant`, потому что хотим, чтобы `Ninject` возвращал имитацию объекта, когда он получает запрос от реализации интерфейса `IProductRepository`:

```
ninjectKernel.Bind<IProductRepository>().ToConstant(mock.Object);
```

Вместо того, чтобы каждый раз создавать новый экземпляр реализации объекта, `Ninject` всегда будет отвечать на запросы интерфейса `IProductRepository` имитацией объекта.

# Отображение списка товаров

Мы могли бы до конца этой главы создавать доменную модель и хранилище, даже не затрагивая проект UI. Но, чтобы вам не стало скучно, мы сменим курс и начнем использовать MVC Framework в полную силу. Мы будем добавлять новые функции в модель и хранилище по мере необходимости.

В этом разделе мы создадим контроллер и метод действия, который будет отображать информацию о товарах, находящихся в хранилище. На данный момент у нас есть данные только в имитированном хранилище, но мы решим этот вопрос позже. Мы также проведем начальные настройки *конфигурации маршрутизации*, чтобы MVC было известно, как отображать запросы к приложению в контроллер, который мы собираемся создать.

## Добавляем контроллер

Кликните правой кнопкой мыши папку `Controllers` в проекте `SportsStore.WebUI` и выберите `Add - Controller` в контекстном меню. Назначьте контроллеру имя `ProductController` и убедитесь, что опция `Template` содержит `Empty controller`. Когда Visual Studio открывает файл для редактирования, вы можете удалить метод действия по умолчанию, который был добавлен автоматически, и ваш файл будет выглядеть как в листинге 7-6.

### Листинг 7-6: Начальное определение `ProductController`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;

namespace SportsStore.WebUI.Controllers
{
    public class ProductController : Controller
    {
        private IProductRepository repository;
        public ProductController(IProductRepository productRepository)
        {
            this.repository = productRepository;
        }
    }
}
```

Удалив метод действия `Index`, мы добавляем конструктор, который принимает параметр `IProductRepository`. Это позволит `Ninject` внедрять зависимость для хранилища товаров, когда он будет создавать экземпляр класса контроллера. Мы также импортировали пространства имен `SportsStore.Domain`, так что мы можем обращаться к хранилищу и классам моделей, не указывая их имен.

Далее мы добавляем метод действия под названием `List`, который будет визуализировать представление, показывающее полный список товаров, как показано в листинге 7-7.

### Листинг 7-7: Добавляем метод действия

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
```

```

using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;

namespace SportsStore.WebUI.Controllers
{
    public class ProductController : Controller
    {
        private IProductRepository repository;
        public ProductController(IProductRepository productRepository)
        {
            this.repository = productRepository;
        }

        public ViewResult List()
        {
            return View(repository.Products);
        }
    }
}

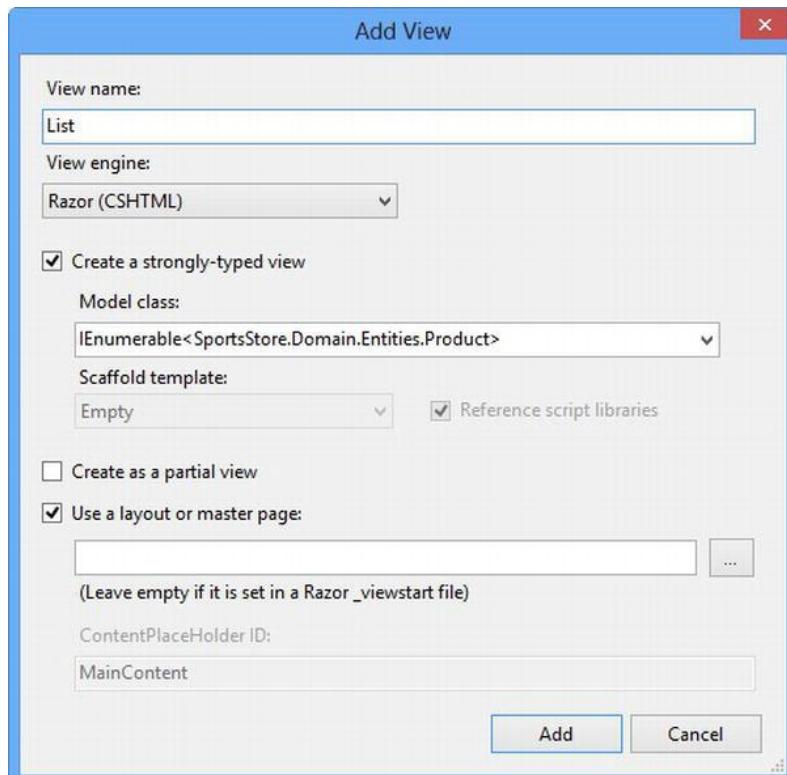
```

Такой вызов метода `View` (без указания имени представления) сообщает платформе визуализировать представление по умолчанию для данного метода действия. Передавая список `List` объектов `Product` в метод `View`, мы предоставляем платформе данные для заполнения объекта `Model` в строго типизированном представлении.

## Добавляем представление

Теперь нам нужно добавить представление по умолчанию для метода действия `List`. Щелкните правой кнопкой мыши метод `List` в редакторе кода и выберите `Add - View` в контекстном меню. Присвойте представлению имя `List` и отметьте флажком опцию, которая создает строго типизированные представления, как показано на рисунке 7-5.

**Рисунок 7-5:** Добавляем представление `List`



В поле `Model class` введите `IEnumerable<SportsStore.Domain.Entities.Product>`. Вам придется напечатать это название; оно не будет доступно из раскрывающегося списка, который не включает перечисления доменных объектов.

В дальнейшем мы будем использовать стандартный макет Razor, который включен в шаблон проекта `Basic`, чтобы наши представления выглядели единообразно. Отметьте флажком опцию `Use a layout`, но оставьте текстовое поле пустым, как показано на рисунке. Нажмите кнопку `Add`, чтобы создать представление.

## Визуализируем данные представления

Зная, что модель в представлении является `IEnumerable<Product>`, мы можем создать список с помощью цикла `foreach` в Razor, как показано в листинге 7-8.

**Листинг 7-8:** Представление `List.cshtml`

```
@model IEnumerable<SportsStore.Domain.Entities.Product>

@{
    ViewBag.Title = "Products";
}

@foreach (var p in Model)
{
    <div class="item">
        <h3>@p.Name</h3>
        @p.Description
        <h4>@p.Price.ToString("c")</h4>
    </div>
}
```

Мы также изменили заголовок страницы. Обратите внимание, что нам не нужно использовать элементы Razor `text` или `@:` для отображения данных представления, потому что каждая строка в теле кода либо является директивой Razor, либо начинается с HTML-элемента.

### *Совет*

*Обратите внимание, что мы преобразовали свойство `Price` в строку с помощью метода `ToString("c")`, который отображает числовые значения как валюту в соответствии с настройками культуры, которые действуют на вашем сервере. Например, если сервер настроен как `en-US`, то `(1002,3).ToString("c")` вернет `$1,002.30`, но если сервер настроен как `en-GB`, тот же метод вернет `£1,002,30`. Вы можете изменить настройку культуры для вашего сервера, добавив в узел `<system.web>` файла `Web.config` следующую секцию: `<globalization culture="en-GB" uiCulture="en-GB" />`.*

## Настраиваем роут по умолчанию

Сейчас нам достаточно сообщить платформе MVC, что запросы, поступающие в корень сайта (`http://mysite/`), нужно отображать в метод действия `List` класса `ProductController`. Для этого мы редактируем оператор в методе `RegisterRoutes` в файле `App_Start\RouteConfig.cs`, как показано в листинге 7-9.

### Листинг 7-9: Добавляем роут по умолчанию

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace SportsStore.WebUI
{
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new
                {
                    controller = "Product",
                    action = "List",
                    id = UrlParameter.Optional
                }
            );
        }
    }
}
```

Измените `Home` на `Product` и `Index` на `List`, как показано в листинге. Мы подробно опишем возможности маршрутизации в ASP.NET в главе 13. На данный момент достаточно знать, что это изменение будет направлять запросы к URL по умолчанию в метод действия, который мы определили.

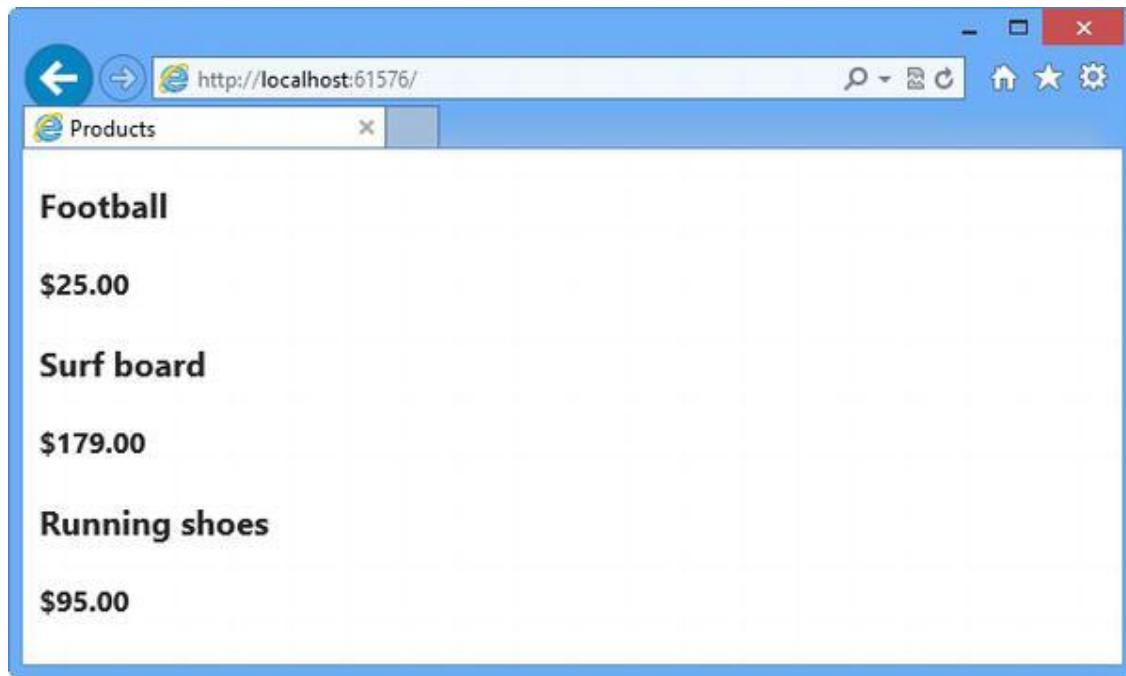
#### *Совет*

*Обратите внимание, что в листинге 7-9 мы установили контроллеру значение `Product`, а не `ProductController`, что является именем класса. Это часть схемы именования ASP.NET MVC, в которой имена классов контроллеров всегда заканчиваются на `Controller`, и при обращении к классу эта часть имени опускается.*

## Запускаем приложение

Все базовые составляющие приложения готовы. У нас есть контроллер с методом действия, который вызывается при запросе URL по умолчанию. Этот метод действия полагается на имитированную реализацию нашего интерфейса хранилища, которая генерирует простые тестовые данные. Тестовые данные передаются в представление, которое мы связали с методом действия, и оно создает простой список с информацией для каждого товара. Если вы запустите приложение, то увидите результат, показанный на рисунке 7-6.

Рисунок 7-6: Просматриваем базовую функциональность приложения



Это типичный шаблон разработки для платформы ASP.NET MVC. Мы отводим довольно много времени на настройку, но затем разработка базовой функциональности приложения происходит очень быстро.

## Подготовка базы данных

Мы уже можем отображать простые представления, содержащие информацию о товарах, но мы по-прежнему выводим тестовые данные, которые возвращает наше имитированное хранилище `IProductRepository`. Прежде чем мы сможем реализовать реальное хранилище, нам нужно создать базу данных и заполнить ее данными.

В качестве базы данных мы будем использовать SQL Server и обращаться к ней с помощью Entity Framework (EF), которая является ORM-платформой .NET. Платформа ORM позволяет нам работать с таблицами, столбцами и строками в реляционной базе данных с помощью обычных объектов C#. Мы уже упоминали в главе 6, что LINQ может работать с различными источниками данных, одним из которых является Entity Framework. Вы вскоре увидите, как она упрощает работу.

Это еще один источник, который предоставляет нам широкий спектр инструментов и технологий. Здесь доступны не только различные реляционные базы данных, но вы также можете работать с хранилищами объектов, архивами документов и некоторыми малоизвестными альтернативами. Существует много платформ ORM, в которых применяются немного разные подходы и которые будут лучше подходить для тех или иных проектов.

Мы используем Entity Framework по нескольким причинам. Во-первых, ее легко настроить и начать работу. Во-вторых, мы хотим использовать LINQ, а у нее первоклассная интеграция с LINQ. Третья причина заключается в том, что Entity Framework на самом деле очень хорошая платформа. Более ранние версии были немного неточными, но современные версии многофункциональны и отлично работают.

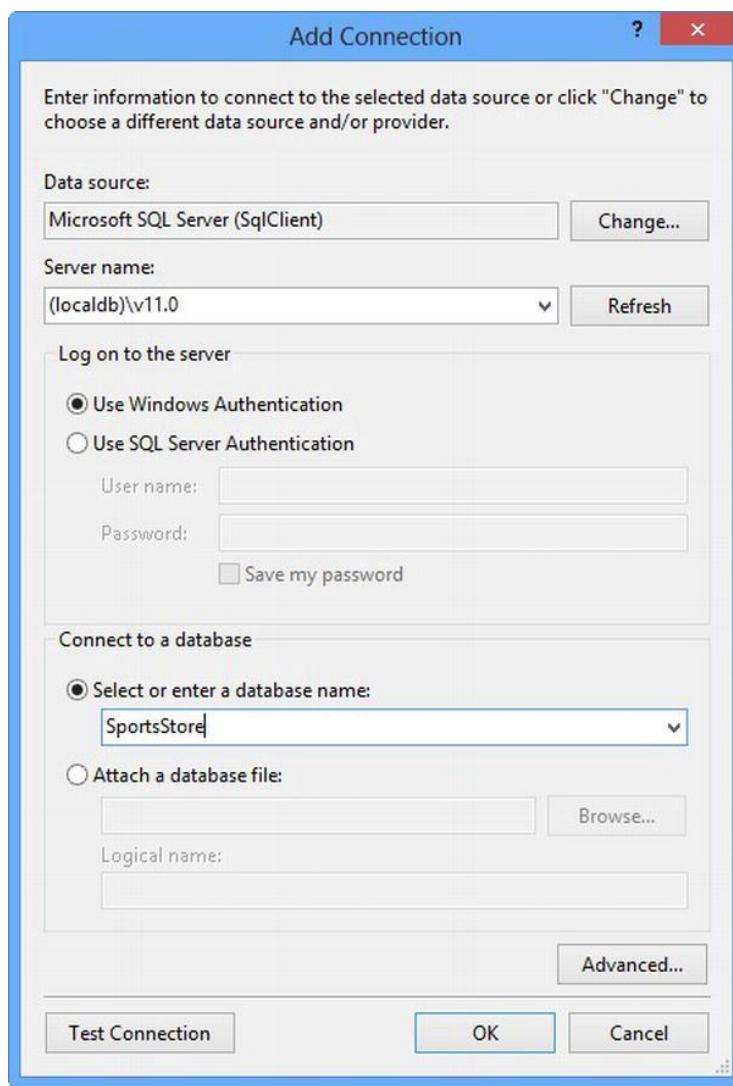
## Создаем базу данных

Одно из приятных дополнений к Visual Studio 2012 и SQL Server 2012 - версия LocalDB. Это безадминистративная реализация функций ядра SQL Server, предназначенных только для разработчиков. Используя эту версию, мы можем пропустить процесс настройки базы данных, пока мы создаем проект, а затем развернуть приложение на полном экземпляре SQL Server. Большинство приложений MVC развертываются на удаленных платформах, которые находятся в ведении профессиональных администраторов. Версия LocalDB означает, что конфигурацию базы данных можно оставить администраторам баз данных, а разработчики смогут заняться кодированием. Версия LocalDB автоматически устанавливается вместе с Visual Studio Express 2012 for Web, но при желании вы можете скачать ее непосредственно с сайта [www.microsoft.com/SQLServer](http://www.microsoft.com/SQLServer).

Для начала мы создадим соединение с базой данных в Visual Studio. Откройте Database Explorer в меню View и кликните по кнопке Connect to Database (которая выглядит как кабель питания с зеленым плюсом).

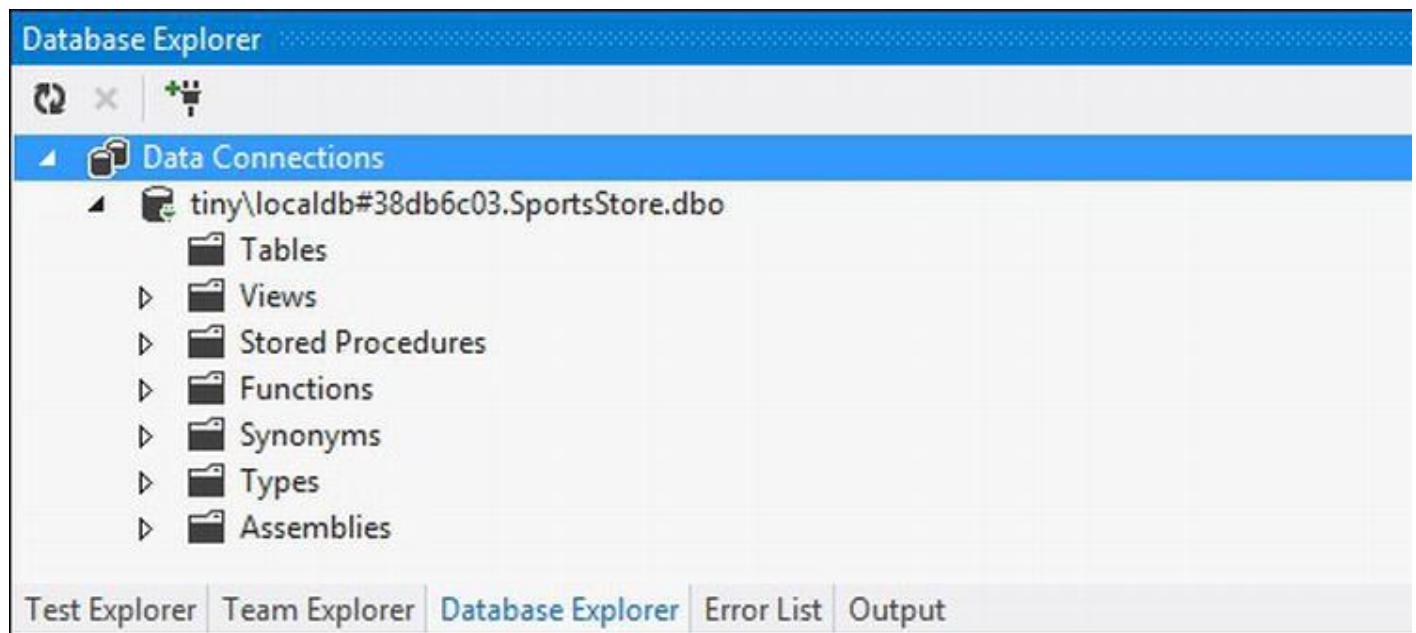
Вы увидите диалоговое окно Add Connection. Присвойте серверу имя (localdb)\v11.0 - это специальное имя, которое означает, что вы хотите использовать версию LocalDB. Отметьте флажком опцию Use Windows Authentication и назовите базу данных SportsStore, как показано на рисунке 7-7.

Рисунок 7-7: Настраиваем базу данных SportsStore



Нажмите OK, и вам будет предложено создать новую базу данных. Нажмите Yes, и в окне Database Explorer появится новая запись. Разверните этот пункт, чтобы увидеть различные аспекты новой базы данных, как показано на рисунке 7-8.

Рисунок 7-8: База данных LocalDB в окне Database Explorer

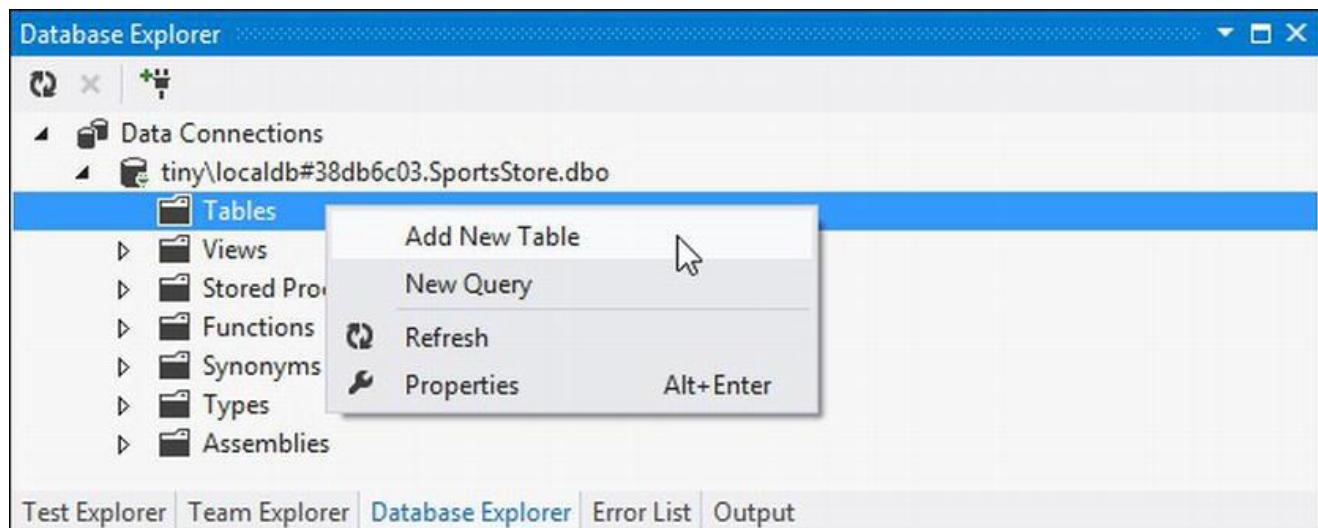


Вы должны увидеть что-то очень похожее, только название соединения с базой данных будет другим, хотя бы потому, что оно включает имя локального компьютера (наш называется tiny).

### Определяем схему базы данных

В нашей базе данных нам нужна только одна таблица, в которой мы будем хранить объекты Product. В окне Database Explorer разверните базу данных, которую вы только что создали, и кликните правой кнопкой мыши по пункту Tables. Выберите из меню Add - New Table, как показано на рисунке 7-9.

Рисунок 7-9: Добавляем новую таблицу



Вы увидите конструктор для создания новой таблицы. Вы можете создавать новые таблицы базы данных с помощью визуальной части конструктора, но мы будем использовать окно T-SQL, потому что это более краткий и точный способ описать необходимую спецификацию таблицы. Введите SQL-оператор, показанный в листинге 7-10, и нажмите кнопку Update в верхнем левом углу окна конструктора.

**Листинг 7-10:** SQL-оператор для создания таблицы в базе данных SportsStore

```
CREATE TABLE Products
(
    [ProductID] INT NOT NULL PRIMARY KEY IDENTITY,
    [Name] NVARCHAR(100) NOT NULL,
    [Description] NVARCHAR(500) NOT NULL,
    [Category] NVARCHAR(50) NOT NULL,
    [Price] DECIMAL(16, 2) NOT NULL
)
```

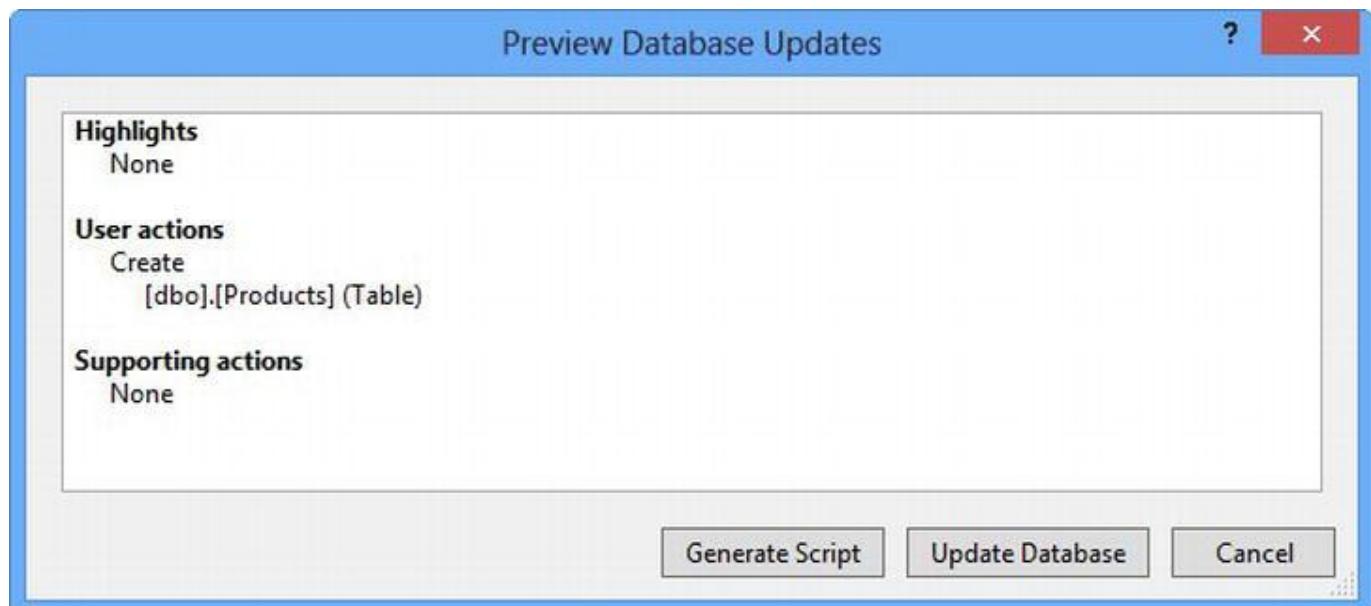
Этот оператор создает таблицу под названием `Products`, которая содержит столбцы для различных свойств, которые мы определили в классе модели `Product` ранее в этой главе.

*Совет*

*Настройка свойства `IDENTITY` для столбца `ProductID` означает, что SQL Server будет генерировать уникальный первичный ключ, когда мы добавляем данные в эту таблицу. Когда мы используем базу данных в приложении, будет очень трудно генерировать уникальные первичные ключи, потому что запросы от пользователей приходят одновременно. Если мы включим эту функцию, то сможем сохранять новые строки таблицы, причем назначать уникальные ключи будет SQL Server.*

Когда вы нажмете кнопку `Update`, вам будет показана информация об эффекте оператора, как показано на рисунке 7-10.

**Рисунок 7-10:** Информация об эффекте SQL-оператора



Нажмите кнопку `Update Database`, чтобы выполнить SQL и создать таблицу `Products` в базе данных. Вы сможете увидеть эффект обновления, если нажмете на кнопку `Refresh` в окне `Database Explorer`. Раздел `Tables` показывает новую таблицу `Product` и информацию о каждой строке.

## Совет

После обновления базы данных вы можете закрыть окно `dbo.Products`. Visual Studio предложит сохранить SQL-скрипт, который вы использовали для создания базы данных. Вам не нужно его сохранять в этой главе, но это может быть полезно сделать в реальных проектах, если вам понадобится настраивать несколько баз данных.

## Добавляем данные в базу данных

Мы добавим в базу некоторые данные вручную, чтобы было с чем работать до тех пор, пока не добавим средства администрирования каталога в главе 10.

В окне `Database Explorer` разверните пункт `Tables` базы данных `SportsStore` кликните правой кнопкой мыши таблицу `Products` и выберите `Show - Table Data`. Введите данные, показанные на рисунке 7-11. Переходить на другую строку можно с помощью клавиши `Tab`. Нажимая `Tab` в конце каждой строки, вы перейдете к следующей строке и обновите данные в базе.

## Внимание

Вы должны оставить столбец `ProductID` пустым. Это столбец идентификации, для которого SQL Server создаст уникальное значение, когда вы перейдете на другую строку.

Рисунок 7-11: Добавляем данные в таблицу `Products`

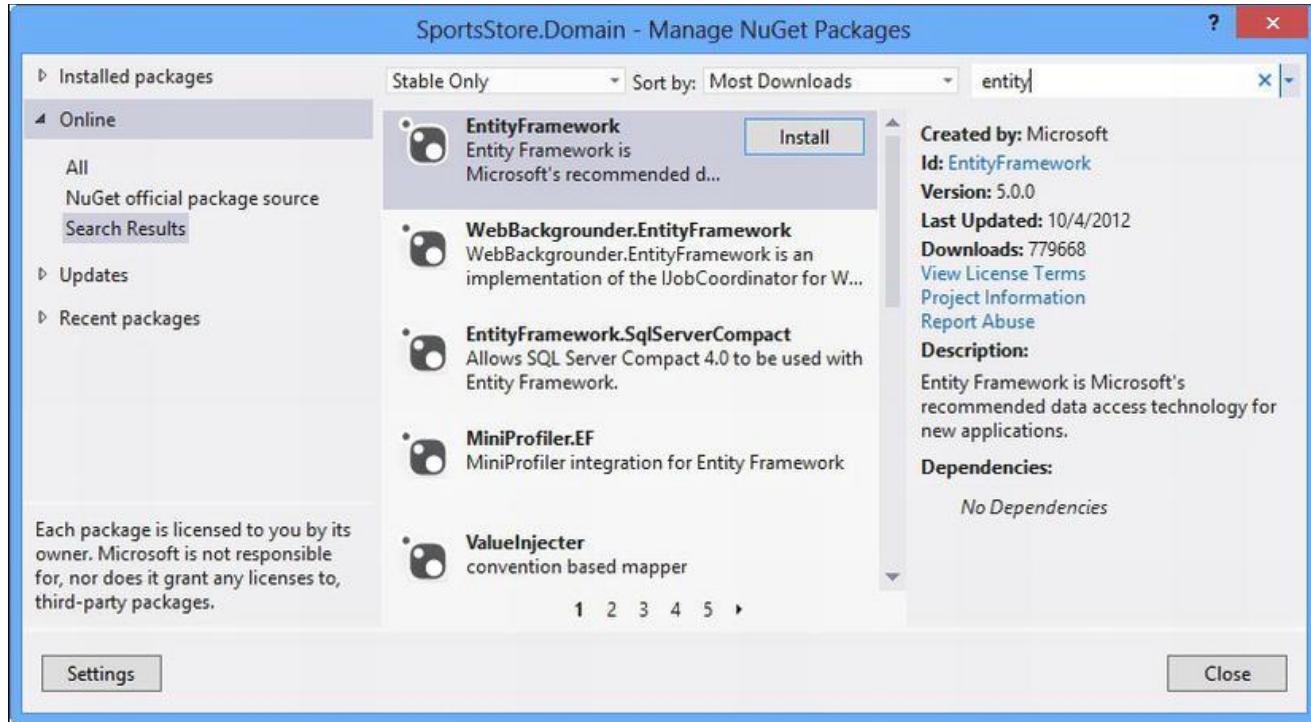
	ProductID	Name	Description	Category	Price
1		Kayak	A boat for one person	Watersports	275.00
2		Lifejacket	Protective and fashionable	Watersports	48.95
3		Soccer Ball	FIFA-approved size and weight	Soccer	19.50
4		Corner Flags	Give your playing field a professional touch	Soccer	34.95
5		Stadium	Flat-packed 35,000-seat stadium	Soccer	79500.00
6		Thinking Cap	Improve your brain efficiency by 75%	Chess	16.00
7		Unsteady Chair	Secretly give your opponent a disadvantage	Chess	29.95
9		Human Chess Board	A fun game for the family	Chess	75.00
10		Bling-Bling King	Gold-plated, diamond-studded King	Chess	1200.00
*	NULL	NULL	NULL	NULL	NULL

## Создаем контекст Entity Framework

Последние версии Entity Framework включает отличную возможность под названием *code-first*. Смысл том, что мы можем определить классы в нашей модели, а затем сгенерировать базы данных из этих классов.

Это очень удобно для проектов, которые разрабатываются с нуля. Но так как таких проектов очень мало, мы продемонстрируем вам версию code-first, в которой будем связывать классы моделей с существующей базой данных. Для начала мы добавим версию Entity Framework 5.0 в проект SportsStore.Domain. Кликните правой кнопкой мыши пункт References в Solution Explorer, а затем выберите пункт Manage NuGet Packages в контекстном меню. Введите entity в поле поиска и выберите поиск пакета EntityFramework, как показано на рисунке 7-12, а затем нажмите кнопку Install. Visual Studio скачает и установит последнюю версию пакета Entity Framework.

**Рисунок 7-12:** Добавляем библиотечный пакет EntityFramework



Следующим шагом будет создание класса контекста, который будет связывать нашу простую модель с базой данных. Создайте новую папку под названием Concrete и добавьте в нее класс под названием EFDbContext. Отредактируйте содержимое файла так, как показано в листинге 7-11.

**Листинг 7-11:** Класс EFDbContext

```
using SportsStore.Domain.Entities;
using System.Data.Entity;

namespace SportsStore.Domain.Concrete
{
    public class EFDbContext : DbContext
    {
        public DbSet<Product> Products { get; set; }
    }
}
```

Чтобы воспользоваться возможностью code-first, мы должны создать класс, который наследует от System.Data.Entity.DbContext. Этот класс будет автоматически определять свойство для каждой таблицы в базе данных, с которой мы будем работать.

Имя свойства указывает таблицу, а параметр типа результата DbSet конкретизирует модель, которую должна использовать Entity Framework для представления строк в данной таблице. В нашем случае

свойство называется `Products`, тип - `Product`. Для репрезентации строк в таблице `Products` мы хотим использовать тип модели `Product`.

Мы должны сообщить Entity Framework, как подключаться к базе данных, и для этого мы добавляем строку подключения к базе данных в файл `Web.config` в проекте `SportsStore.WebUI`, которая имеет то же имя, что и класс контекста, как показано в листинге 7-12.

**Совет**

*Обратите внимание, что здесь мы перешли на другой проект. Мы определяем модель и логику хранения данных в проекте `SportsStore.Domain`, но информацию о подключении к базе данных помещаем в файле `Web.config` в проекте `SportsStore.WebUI`.*

**Листинг 7-12:** Добавляем подключение к базе данных

```
<connectionStrings>
  <add name="EFDbContext"
    connectionString="Data Source=(localdb)\v11.0;Initial
Catalog=SportsStore;Integrated Security=True"
    providerName="System.Data.SqlClient"/>
</connectionStrings>
```

В разделе `connectionStrings` файла `Web.config` будет еще один элемент `add`, который Visual Studio создает по умолчанию. Вы можете либо игнорировать его, либо, как мы, удалить.

## Создаем хранилище Product

Теперь у нас есть все, что нужно, чтобы реализовать реальный класс `IProductRepository`. Добавьте класс под названием `EFProductRepository` в папку `Concrete` проекта `SportsStore.Domain`. Измените файл класса так, чтобы он соответствовал листингу 7-13.

**Листинг 7-13:** `EFProductRepository.cs`

```
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using System.Linq;

namespace SportsStore.Domain.Concrete
{
    public class EFProductRepository : IProductRepository
    {
        private EFDbContext context = new EFDbContext();
        public IQueryable<Product> Products
        {
            get { return context.Products; }
        }
    }
}
```

Это наш класс хранилища. Он реализует интерфейс `IProductRepository` и использует экземпляр `EFDbContext`, чтобы извлекать данные из базы с помощью Entity Framework. Мы продемонстрируем приемы работы с Entity Framework (и насколько они простые), когда будем добавлять в хранилище новые функции.

Наконец мы заменим привязку Ninject к нашему имитированному хранилищу на привязку к реальному хранилищу. Отредактируйте класс `NinjectControllerFactory` в проекте `SportsStore.WebUI` так, чтобы метод `AddBindings` соответствовал листингу 7-14.

**Листинг 7-14:** Добавляем привязку к реальному хранилищу

```
using System;
using System.Web.Mvc;
using System.Web.Routing;
using Ninject;
using SportsStore.Domain.Entities;
using SportsStore.Domain.Abstract;
using System.Collections.Generic;
using System.Linq;
using Moq;
using SportsStore.Domain.Concrete;

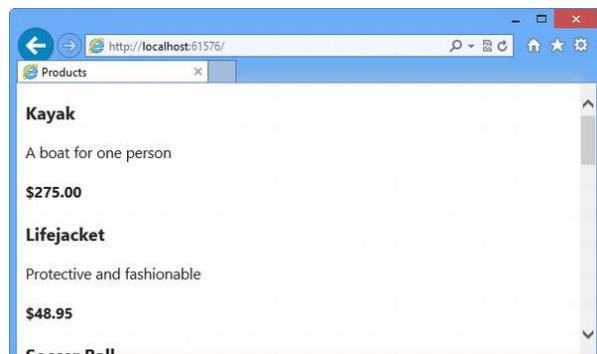
namespace SportsStore.WebUI.Infrastructure
{
    public class NinjectControllerFactory : DefaultControllerFactory
    {
        private IKernel ninjectKernel;
        public NinjectControllerFactory()
        {
            ninjectKernel = new StandardKernel();
            AddBindings();
        }

        protected override IController GetControllerInstance(RequestContext requestContext,
Type controllerType)
        {
            return controllerType == null
                ? null
                : (IController)ninjectKernel.Get(controllerType);
        }

        private void AddBindings()
        {
            ninjectKernel.Bind<IProductRepository>().To<EFProductRepository>();
        }
    }
}
```

Новая привязка выделена жирным шрифтом. Она сообщает Ninject, что мы хотим создавать экземпляры класса `EFProductRepository` для обслуживания запросов к интерфейсу `IProductRepository`. Теперь остается только запустить приложение еще раз. Результаты показаны на рисунке 7-13, и вы можете видеть, что сейчас наш список содержит данные о товарах, которые мы занесли в базу данных.

**Рисунок 7-13:** Результат реализации реального хранилища



Этот подход использования Entity Framework для представления базы данных SQL Server как серии объектов модели - очень простой, и он позволяет нам оставаться сфокусированными на MVC Framework. Конечно, мы пропускаем много деталей касательно того, как работает Entity Framework, и огромное количество доступных опций конфигурации. Нам действительно нравится Entity Framework, и мы рекомендуем вам потратить немного времени и познакомиться с ним более подробно. Начать можно с сайта Microsoft для Entity Framework: <http://msdn.microsoft.com/data/ef>.

## Добавление нумерации страниц

Как вы можете видеть на рисунке 7-13, все товары из базы данных отображаются на одной странице. В этом разделе мы добавим поддержку нумерации страниц, чтобы на одной странице отображать определенное количество товаров, и чтобы пользователь мог бы посматривать каталог, переходя с одной страницы на другую. Для этого мы добавим в метод `List` контроллера `Product` параметр, показанный в листинге 7-15.

**Листинг 7-15:** Добавляем поддержку нумерации страниц в метод `List` контроллера `Product`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;

namespace SportsStore.WebUI.Controllers
{
    public class ProductController : Controller
    {
        private IProductRepository repository;
        public int PageSize = 4;
        public ProductController(IProductRepository productRepository)
        {
            this.repository = productRepository;
        }

        public ViewResult List(int page = 1)
        {
            return View(repository.Products
                .OrderBy(p => p.ProductID)
                .Skip((page - 1) * PageSize)
                .Take(PageSize));
        }
    }
}
```

Дополнения в классе контроллера выделены жирным шрифтом. Поле `PageSize` указывает, что мы хотим видеть четыре товара на странице. В дальнейшем мы вернемся к этому механизму и заменим его на более действенный. Мы добавили дополнительный параметр в метод `List`. Это означает, что при вызове метода без параметра (`List ()`) наш вызов обрабатывается так, как если бы мы указали значение параметра (`List (1)`). В результате, если мы не указываем номер страницы, мы получим первую страницу. LINQ делает нумерацию страниц очень простой. В методе `List` мы получаем объекты `Product` из хранилища, упорядочиваем их по первичному ключу, пропускаем товары, которые идут до начала нашей страницы, и берем количество товаров, указанное в поле `PageSize`.

## Модульный тест: нумерация страниц

Чтобы протестировать функцию нумерации страниц, мы можем создать имитированное хранилище, внедрить его в конструктор класса `ProductController`, а затем вызвать метод `List` и запросить конкретную страницу. Далее мы можем сравнить те объекты `Product`, которые получили, с теми, которые ожидали получить из тестовых данных в имитированной реализации. Подробно создание модульных тестов описано в главе 6. Вот тест, который мы создали для этой цели:

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using SportsStore.WebUI.Controllers;
using System.Collections.Generic;
using System.Linq;

namespace SportsStore.UnitTests
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void Can_Paginate()
        {
            // Arrange
            Mock<IProductRepository> mock = new Mock<IProductRepository>();
            mock.Setup(m => m.Products).Returns(new Product[] {
                new Product { ProductID = 1, Name = "P1" },
                new Product { ProductID = 2, Name = "P2" },
                new Product { ProductID = 3, Name = "P3" },
                new Product { ProductID = 4, Name = "P4" },
                new Product { ProductID = 5, Name = "P5" }
            }.AsQueryable());

            ProductController controller = new ProductController(mock.Object);

            controller.PageSize = 3;

            // Act
            IEnumerable<Product> result = (IEnumerable<Product>)controller.List(2).Model;

            // Assert
            Product[] prodArray = result.ToArray();
            Assert.IsTrue(prodArray.Length == 2);
            Assert.AreEqual(prodArray[0].Name, "P4");
            Assert.AreEqual(prodArray[1].Name, "P5");
        }
    }
}
```

Обратите внимание, как легко можно получить данные, возвращаемые из метода контроллера. Мы вызываем свойство `Model` в результате, чтобы получить последовательность `IEnumerable<Product>`, которую мы генерировали в методе `List`. Затем мы можем проверить, те ли это данные, которые мы хотим получить. В этом случае мы преобразовали последовательность в массив, и проверили длину и значение отдельных объектов.

## Отображаем ссылки на страницы

Если вы запустите приложение, то увидите, что на странице отображаются только четыре элемента. Если вы хотите просмотреть другую страницу, можно добавить параметры строки запроса в конец URL, например:

```
http://localhost:23081/?page=2
```

Вам нужно будет изменить порт в URL, чтобы он соответствовал порту вашего рабочего сервера ASP.NET. С помощью этих строк запросов мы можем просмотреть весь каталог товаров.

Конечно, этот способ известен только нам. Клиенты не знают, что можно использовать параметры строк запросов, и, даже если знают, мы все равно уверены, что они не собираются просматривать каталог таким образом. Нам нужно отображать ссылки на страницы под каждым списком товаров, чтобы клиенты могли переходить с одной страницы на другую. Для этого мы реализуем многоразовый вспомогательный метод HTML, похожий на методы `Html.TextBoxFor` и `Html.BeginForm`, с которыми мы работали в главе 2. Этот вспомогательный метод будет генерировать разметку HTML для необходимых нам навигационных ссылок.

## Добавляем модель представления

Для поддержки вспомогательного метода HTML, мы будем передавать в представление информацию о количестве доступных страниц, текущей странице и общем количестве товаров в хранилище. Самый простой способ это сделать - создать модель представления, о которой мы кратко упоминали в главе 3. Добавьте класс под названием `PagingInfo`, показанный в листинге 7-16, в папку в `Models` в проекте `SportsStore.WebUI`.

### Листинг 7-16: Класс модели представления `PagingInfo`

```
using System;

namespace SportsStore.WebUI.Models
{
    public class PagingInfo
    {
        public int TotalItems { get; set; }
        public int ItemsPerPage { get; set; }
        public int CurrentPage { get; set; }

        public int TotalPages
        {
            get { return (int)Math.Ceiling((decimal)TotalItems / ItemsPerPage); }
        }
    }
}
```

Модель представления не является частью доменной модели. Это просто удобный класс для передачи данных между представлением и контроллером. Чтобы подчеркнуть это, мы помещаем этот класс в проект `SportsStore.WebUI`, чтобы держать его отдельно от классов доменной модели.

## Добавляем вспомогательный метод HTML

Теперь, когда у нас есть модель представления, мы реализуем вспомогательный метод HTML, который назовем `PageLinks`. Создайте в проекте `SportsStore.WebUI` новую папку под названием `HtmlHelpers`, и добавьте новый статический класс под названием `PagingHelpers`. Содержимое файла класса показано в листинге 7-17.

### Листинг 7-17: Класс PagingHelpers

```
using System;
using System.Text;
using System.Web.Mvc;
using SportsStore.WebUI.Models;

namespace SportsStore.WebUI.HtmlHelpers
{
    public static class PagingHelpers
    {
        public static MvcHtmlString PageLinks(
            this HtmlHelper html,
            PagingInfo pagingInfo,
            Func<int, string> pageUrl)
        {
            StringBuilder result = new StringBuilder();

            for (int i = 1; i <= pagingInfo.TotalPages; i++)
            {
                TagBuilder tag = new TagBuilder("a"); // Construct an <a> tag
                tag.MergeAttribute("href", pageUrl(i));
                tag.InnerHtml = i.ToString();
                if (i == pagingInfo.CurrentPage)
                    tag.AddCssClass("selected");
                result.Append(tag.ToString());
            }

            return MvcHtmlString.Create(result.ToString());
        }
    }
}
```

Метод расширения `PageLinks` генерирует HTML для набора ссылок на страницы, используя информацию, предоставленную в объекте `PagingInfo`. Параметр `Func` предоставляет возможность **передачи делегата**, который будет использоваться для генерации ссылок на другие страницы.

### Модульный тест: создание ссылок на страницы

Для проверки вспомогательного метода `PageLinks` мы вызываем метод с тестовыми данными и сравниваем результаты с ожидаемым HTML. Метод модульного теста выглядит следующим образом:

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using SportsStore.WebUI.Controllers;
using SportsStore.WebUI.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.Mvc;
using SportsStore.WebUI.HtmlHelpers;

namespace SportsStore.UnitTests
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void Can_Paginate()
        {
```

```
// ...statements removed for brevity...
}

[TestMethod]
public void Can_Generate_Page_Links()
{
    // Arrange - define an HTML helper - we need to do this
    // in order to apply the extension method
    HtmlHelper myHelper = null;

    // Arrange - create PagingInfo data
    PagingInfo pagingInfo = new PagingInfo
    {
        CurrentPage = 2,
        TotalItems = 28,
        ItemsPerPage = 10
    };

    // Arrange - set up the delegate using a lambda expression
    Func<int, string> pageUrlDelegate = i => "Page" + i;

    // Act
    MvcHtmlString result = myHelper.PageLinks(pagingInfo, pageUrlDelegate);

    // Assert
    Assert.AreEqual(result.ToString(), @"<a href=""Page1"">1</a>" +
        + @"<a class=""selected"" href=""Page2"">2</a>" +
        + @"<a href=""Page3"">3</a>");
}
}
```

Этот тест проверяет вывод вспомогательного метода, используя значение литеральной строки, которая содержит двойные кавычки. C# прекрасно работает с такими строками до тех пор, пока мы ставим перед строкой @ и используем два набора двойных кавычек ("") вместо одного набора. Мы должны также помнить, что нельзя разбивать литеральную строку на отдельные строки, если только строка, с которой мы сравниваем, не разбита аналогично. Так, например, литерал, который мы используем в тестовом методе, занял две строки из-за небольшой ширины страницы. Мы не добавили символ новой строки; если бы мы это сделали, тест завершился бы неудачей.

Метод расширения доступен для использования только тогда, когда содержащее его пространство имен находится в области видимости. В файле кода это делается с помощью оператора `using`; в представлении Razor мы должны добавить запись конфигурации в файл `Web.config`, или оператор `@using` в само представление. Это может привести к путанице, но в проекте MVC Razor есть два файла `Web.config`: основной, который находится в корневой директории проекта приложения, и специальный для представлений, который находится в папке `Views`. Данное изменение мы должны провести в файле `Views/Web.config`, и оно показано в листинге 7-18.

**Листинг 7-18:** Добавляем пространство имен вспомогательного метода HTML в файл Views/Web.config

```
<system.web.webPages.razor>
  <host factoryType="System.Web.Mvc.MvcWebRazorHostFactory, System.Web.Mvc,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35" />
  <pages pageBaseType="System.Web.Mvc.WebViewPage">
    <namespaces>
      <add namespace="System.Web.Mvc" />
      <add namespace="System.Web.Mvc.Ajax" />
      <add namespace="System.Web.Mvc.Html" />
      <add namespace="System.Web.Optimization"/>
      <add namespace="System.Web.Routing" />
```

```

<add namespace="SportsStore.WebUI.HtmlHelpers"/>
</namespaces>
</pages>
</system.web.webPages.razor>

```

Каждое пространство имен, к которому нам потребуется обратиться в представлении Razor, нужно объявить либо таким образом, либо в самом представлении с помощью оператора @using.

## Добавляем данные модели представления

Мы еще не вполне готовы использовать наш вспомогательный метод HTML. Нам осталось предоставить экземпляр класса модели представления PagingInfo в представление. Это можно было бы сделать, используя ViewBag, но лучше мы объединим все данные, которые мы собираемся отправить из контроллера в представление, в один класс модели представления. Для этого добавьте новый класс под названием ProductsListViewModel в папку Models проекта SportsStore.WebUI. Содержание этого класса показано в листинге 7-19.

### Листинг 7-19: Класс ProductsListViewModel

```

using System.Collections.Generic;
using SportsStore.Domain.Entities;

namespace SportsStore.WebUI.Models
{
    public class ProductsListViewModel
    {
        public IEnumerable<Product> Products { get; set; }
        public PagingInfo PagingInfo { get; set; }
    }
}

```

Теперь мы можем обновить метод List в классе ProductController так, чтобы он начал использовать класс ProductsListViewModel и предоставлять представлению сведения о товарах, которые нужно отображать на странице, и информацию о нумерации страниц, как показано в листинге 7-20.

### Листинг 7-20: Обновляем метод List

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using SportsStore.WebUI.Models;

namespace SportsStore.WebUI.Controllers
{
    public class ProductController : Controller
    {
        private IProductRepository repository;
        public int PageSize = 4;

        public ProductController(IProductRepository productRepository)
        {
            this.repository = productRepository;
        }

        public ViewResult List(int page = 1)
        {

```

```
ProductsListViewModel model = new ProductsListViewModel
{
    Products = repository.Products
        .OrderBy(p => p.ProductID)
        .Skip((page - 1) * PageSize)
        .Take(PageSize),
    PagingInfo = new PagingInfo
    {
        CurrentPage = page,
        ItemsPerPage = PageSize,
        TotalItems = repository.Products.Count()
    }
};

return View(model);
}
}
```

Эти изменения передают объект ProductsListViewModel в качестве данных модели в представление.

## Модульный тест: данные модели представления о нумерации страниц

Нам нужно убедиться, что контроллер отправляет в представление правильную информацию о нумерации страниц. Чтобы проверить это, мы добавили в наш проект с тестами следующий модульный тест:

```
[TestMethod]
public void Can_Send_Pagination_View_Model()
{
    // Arrange
    Mock<IProductRepository> mock = new Mock<IProductRepository>();
    mock.Setup(m => m.Products).Returns(new Product[] {
        new Product {ProductID = 1, Name = "P1"},
        new Product {ProductID = 2, Name = "P2"},
        new Product {ProductID = 3, Name = "P3"},
        new Product {ProductID = 4, Name = "P4"},
        new Product {ProductID = 5, Name = "P5"}
    }.AsQueryable());

    // Arrange
    ProductController controller = new ProductController(mock.Object);
    controller.PageSize = 3;

    // Act
    ProductsListViewModel result = (ProductsListViewModel)controller.List(2).Model;

    // Assert
    PagingInfo pageInfo = result.PagingInfo;
    Assert.AreEqual(pageInfo.CurrentPage, 2);
    Assert.AreEqual(pageInfo.ItemsPerPage, 3);
    Assert.AreEqual(pageInfo.TotalItems, 5);
    Assert.AreEqual(pageInfo.TotalPages, 2);
}
```

Нам также нужно изменить предыдущий модульный тест для нумерации страниц, который содержится в методе `Can_Paginate`. Он полагается на то, что метод действия `List` возвращает `ViewResult`, у которого свойство `Model` представляет собой последовательность объектов `Product`, но мы заключили данные в другой тип модели представления. Вот обновленный тест:

```
[TestMethod]  
public void Can_Paginate()
```

```

{
    // Arrange
    // - create the mock repository
    Mock<IProductRepository> mock = new Mock<IProductRepository>();
    mock.Setup(m => m.Products).Returns(new Product[] {
        new Product {ProductID = 1, Name = "P1"},
        new Product {ProductID = 2, Name = "P2"},
        new Product {ProductID = 3, Name = "P3"},
        new Product {ProductID = 4, Name = "P4"},
        new Product {ProductID = 5, Name = "P5"}
    }).AsQueryable());

    // create a controller and make the page size 3 items
    ProductController controller = new ProductController(mock.Object);
    controller.PageSize = 3;

    // Action
    ProductsListViewModel result = (ProductsListViewModel)controller.List(2).Model;

    // Assert
    Product[] prodArray = result.Products.ToArray();
    Assert.IsTrue(prodArray.Length == 2);
    Assert.AreEqual(prodArray[0].Name, "P4");
    Assert.AreEqual(prodArray[1].Name, "P5");
}

```

Обычно мы бы создали общий метод настройки, учитывая степень дублирования между этими двумя тестовыми методами. Однако, так как мы выносим модульные тесты в отдельные блоки, такие как этот, мы будем держать весь код отдельно, чтобы вы могли видеть каждый тест целиком.

На данный момент представление ожидает последовательность объектов `Product`, так что нам необходимо обновить `List.cshtml`, как показано в листинге 7-21, чтобы работать с новым типом модели представления.

#### **Листинг 7-21:** Обновляем представление `List.cshtml`

```

@model SportsStore.WebUI.Models.ProductsListViewModel

 @{
    ViewBag.Title = "Products";
}

@foreach (var p in Model.Products)
{
    <div class="item">
        <h3>@p.Name</h3>
        @p.Description
        <h4>@p.Price.ToString("c")</h4>
    </div>
}

```

Мы изменили директиву `@model`, чтобы сообщить Razor, что сейчас мы работаем с другим типом данных. Нам также нужно было обновить цикл `foreach` так, чтобы источником данных стало свойство `Products` данных модели.

#### **Отображаем ссылки на страницы**

Теперь у нас есть все, чтобы добавить ссылки на страницы представления `List`. Мы создали модель представления, которая содержит информацию о нумерации страниц, обновили контроллер так, чтобы эта информация передавалась в представление, и изменили директиву `@model` так, чтобы она

соответствовала новому типу модели представления. Нам осталось только вызвать наш вспомогательный метод HTML из представления, что показано в листинге 7-22.

#### Листинг 7-22: Вызываем вспомогательный метод HTML

```
@model SportsStore.WebUI.Models.ProductsListViewModel

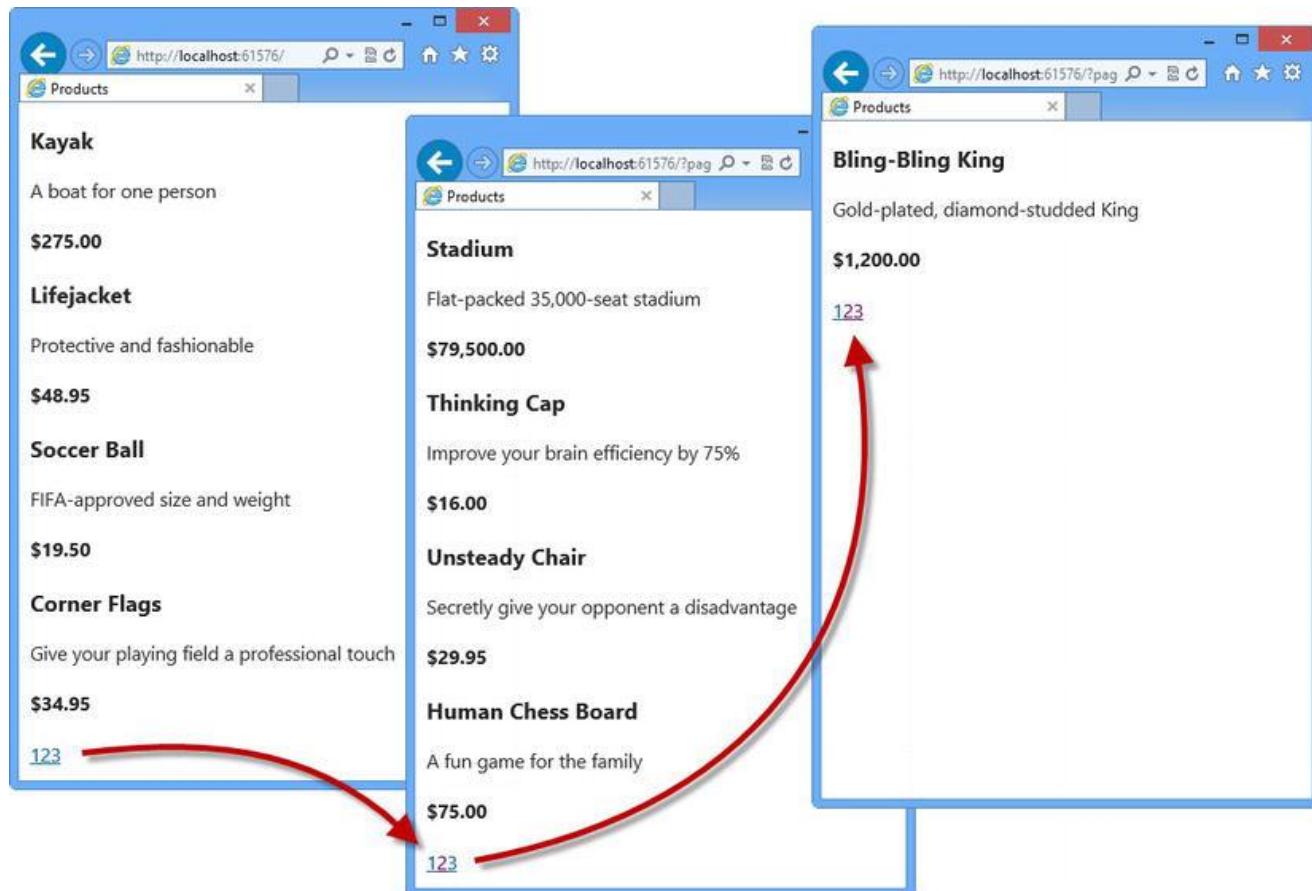
{@{
    ViewBag.Title = "Products";
}

@foreach (var p in Model.Products)
{
    <div class="item">
        <h3>@p.Name</h3>
        @p.Description
        <h4>@p.Price.ToString("c")</h4>
    </div>
}

<div class="pager">
    @Html.PageLinks(Model.PagingInfo, x => Url.Action("List", new { page = x }))
</div>
```

Если вы запустите приложение, вы увидите, что появились ссылки на страницы, как показано на рисунке 7-14. Оформлены они довольно просто, но мы исправим это позже в этой главе. На данный момент важно то, что по ссылкам можно переходить с одной страницы на другую и просматривать товары в каталоге.

Рисунок 7-14: Отображаем навигационные ссылки



## Почему просто не использовать GridView?

Если вы работали раньше с ASP.NET, то можете подумать, что мы выполнили слишком много работы для такого мало впечатляющего результата. Мы создали несколько файлов только для того, чтобы получить нумерацию страниц. Если бы мы использовали Web Forms, мы могли бы сделать то же самое, используя элемент управления ASP.NET Web Forms `GridView` прямо из коробки, просто привязав его к таблице `Products` в базе данных.

То, что мы сделали до сих пор, может не производить глубокого впечатления, но сильно отличается от перетаскивания `GridView` на поверхность разработки. Во-первых, мы строим приложение с надежной и хорошо поддерживаемой архитектурой, которая способствует надлежащему разделению обязанностей. Здесь мы не связываем непосредственно интерфейс с базой данных, в отличие подхода с использованием `GridView`, который дает быстрый результат, но вызывает затруднения и неудобства с течением времени. Во-вторых, в процессе работы мы создаем модульные тесты, которые позволяют нам естественным образом проверять поведение приложения, что практически невозможно при использовании `GridView`.

Наконец, имейте в виду, что значительная часть этой главы посвящена созданию базовой инфраструктуры, на которой строится приложение. Нам необходимо определить и реализовать хранилище, скажем, только один раз, и после этого мы сможем создавать и тестировать новые функции быстро и легко, как продемонстрируют следующие главы.

## Улучшаем URL

У нас уже работают ссылки на страницы, но они все еще используют строки запроса для передачи информации о странице на сервер, к примеру:

```
http://localhost/?page=2
```

Чтобы их улучшить, мы можем создать схему по шаблону компонуемых URL. Компонуемый URL – это ссылка, которая имеет смысл для пользователя, например:

```
http://localhost/Page2
```

К счастью, в MVC очень легко изменять схему URL, поскольку она использует функцию маршрутизации ASP.NET. Для этого нам нужно только добавить новый роут в метод `RegisterRoutes` в файле `RouteConfig.cs`, который вы найдете в папке `App_Start`. Изменения показаны в листинге 7-23.

### Листинг 7-23: Добавляем новый роут

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace SportsStore.WebUI
{
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                "Default", // Name
                "{controller}/{action}/{id}", // URL with parameters
                new { controller = "Home", action = "Index", id = "" } // Parameter defaults
            );
        }
    }
}
```

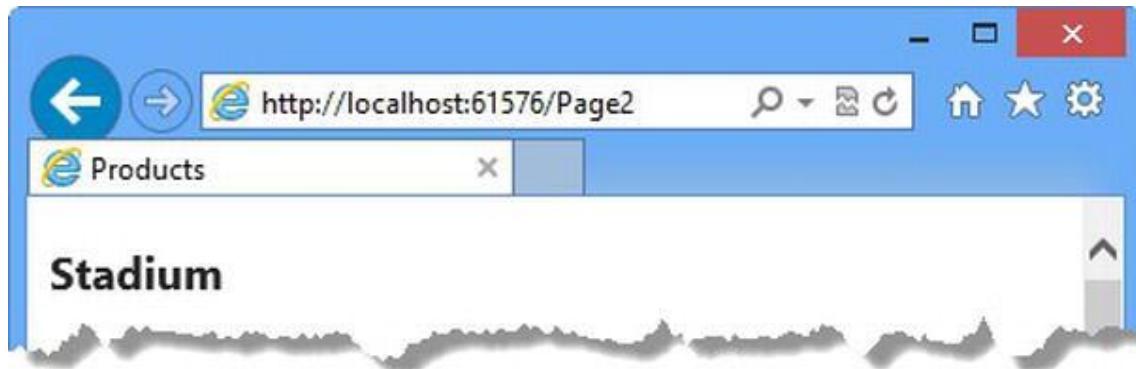
```
        name: null,
        url: "Page{page}",
        defaults: new { Controller = "Product", action = "List" }
    );
}

routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}",
    defaults: new
    {
        controller = "Product",
        action = "List",
        id = UrlParameter.Optional
    }
);
}
}
```

Важно, чтобы вы добавили этот роут перед роутом Default, который уже есть в файле. Как вы увидите в главе 13, роуты обрабатываются в том порядке, в котором они указаны, а нам нужно, чтобы у нового роута был более высокий приоритет, чем у роута по умолчанию.

Это единственное изменение, которое понадобится, чтобы изменить схему URL. MVC Framework тесно интегрирована с функцией маршрутизации, и поэтому изменение вроде этого автоматически отражается в результате выполнения метода `Url.Action` (который мы используем в представлении `List.cshtml` для генерации ссылок на страницы). Если вы совершенно не знакомы с маршрутизацией, не волнуйтесь - мы подробно рассмотрим ее в главе 13. Если вы запустите приложение и перейдите на страницу, вы увидите новую схему URL в действии, как показано на рисунке 7-15.

**Рисунок 7-15:** Новая схема URL отображается в браузере



# Применение стилей к контенту

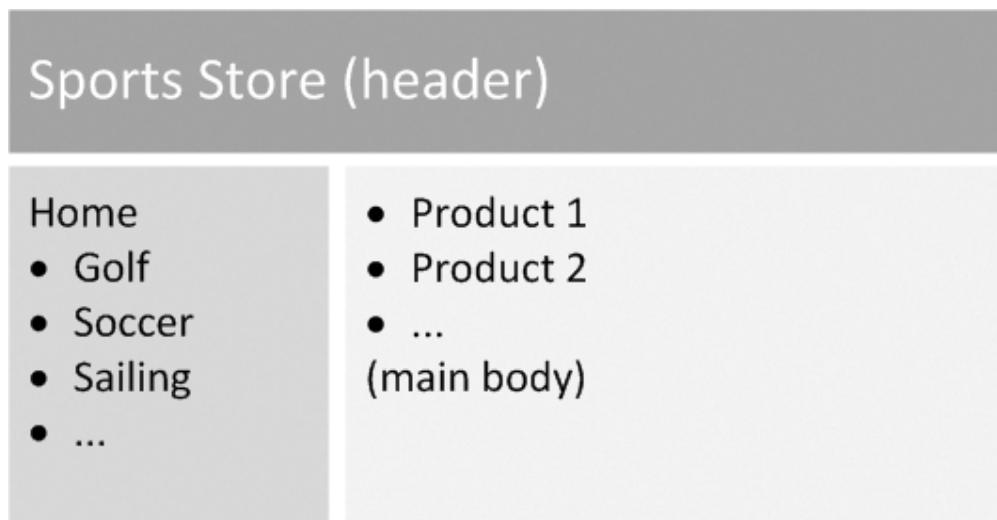
Мы уже создали большую часть инфраструктуры, и наше приложение действительно начинает выглядеть целостным, но мы до сих пор не обратили внимания на его внешний вид. Хотя эта книга не о веб-дизайне или CSS, дизайн приложения SportsStore настолько невыразительный, что умаляет даже его технические достоинства. В данном разделе мы исправим эту ситуацию.

## *Примечание*

*В этой части главы мы попросим вас добавить стили CSS, не объясняя их значение. Адам подробно описывает CSS в своей книге Руководство по HTML5 (Apress, 2011).*

Мы реализуем классический дизайн с двумя колонками и заголовком, как показано на рисунке 7-16.

**Рисунок 7-16:** Целевой дизайн приложения SportsStore



## Определяем общий контент в макете

В главе 5 мы объяснили, как работают макеты Razor и как их применять. Когда мы создавали представление `List.cshtml` для контроллера `Product`, мы попросили вас отметить флагком опцию `Use a layout`, но не заполнять поле, в котором определяется макет. В результате используется макет по умолчанию, `_Layout.cshtml`, который можно найти в папке `Views/Shared` проекта `SportsStore.WebUI`. Откройте этот файл и измените его содержание так, чтобы он соответствовал листингу 7-24.

**Листинг 7-24:** Изменяем стандартный макет Razor

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    <link href="~/Content/Site.css" type="text/css" rel="stylesheet" />
</head>
<body>
    <div id="header">
        <div class="title">SPORTS STORE</div>
    </div>
    <div id="categories">
        We will put something useful here later
    </div>
    <div id="content">
        @RenderBody()
    </div>
</body>
</html>
```

### Примечание

*Обратите внимание, что мы попросили вас удалить теги Razor `@Styles` и `@Scripts` из представления. Это новые дополнения к Razor в MVC 4, и мы опишем их должным образом в главе 24, когда будем говорить о способах оптимизации контента, который ваше приложение MVC демонстрирует клиентам. На данный момент мы хотим сохранить все внимание на основных функциях MVC.*

## Добавляем стили CSS

Разметка HTML в листинге 7-24 является характерной для приложения ASP.NET MVC. Она простая и чисто семантическая. Он описывает контент, но ничего не говорит о том, как он должен быть расположен на экране. Мы будем использовать CSS, чтобы сообщить браузеру, как должны быть расположены элементы, которые мы только что добавили.

Visual Studio создает для нас файл CSS автоматически, если мы создаем проект MVC, используя опцию Basic. Этот файл, который называется Site.css, можно найти в папке Content проекта SportsStore.WebUI, и мы уже просили вас добавить в файл \_Layout.cshtml ссылку на этот файл:

```
<link href="~/Content/Site.css" type="text/css" rel="stylesheet" />
```

Откройте файл Site.css и добавьте стили, показанные в листинге 7-25, в конец файла (не удаляйте содержимое, которое уже есть в Site.css). Вам не нужно набирать их вручную. Вы можете скачать дополнения CSS и весь остальной проект как часть кода, который прилагается к этой книге.

### Листинг 7-25: Определяем CSS

```
BODY
{
    font-family: Cambria, Georgia, "Times New Roman";
    margin: 0;
}

DIV#header DIV.title, DIV.item H3, DIV.item H4, DIV.pager A
{
    font: bold 1em "Arial Narrow", "Franklin Gothic Medium", Arial;
}

DIV#header
{
    background-color: #444;
    border-bottom: 2px solid #111;
    color: White;
}

DIV#header DIV.title
{
    font-size: 2em;
    padding: .6em;
}

DIV#content
{
    border-left: 2px solid gray;
    margin-left: 9em;
    padding: 1em;
}

DIV#categories
{
    float: left;
    width: 8em;
    padding: .3em;
}

DIV.item
{
    border-top: 1px dotted gray;
    padding-top: .7em;
    margin-bottom: .7em;
```

```

}

DIV.item:first-child
{
    border-top: none;
    padding-top: 0;
}

DIV.item H3
{
    font-size: 1.3em;
    margin: 0 0 .25em 0;
}

DIV.item H4
{
    font-size: 1.1em;
    margin: .4em 0 0 0;
}

DIV.pager
{
    text-align: right;
    border-top: 2px solid silver;
    padding: .5em 0 0 0;
    margin-top: 1em;
}

DIV.pager A
{
    font-size: 1.1em;
    color: #666;
    text-decoration: none;
    padding: 0 .4em 0 .4em;
}

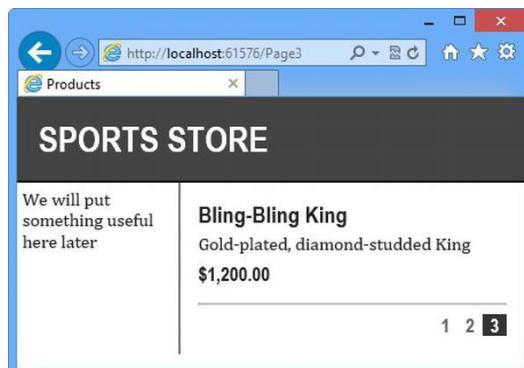
DIV.pager A:hover
{
    background-color: Silver;
}

DIV.pager A.selected
{
    background-color: #353535;
    color: White;
}

```

Если вы запустите приложение, то увидите, что мы уже немного улучшили внешний вид. Изменения показаны на рисунке 7-17.

**Рисунок 7-17:** Приложение SportsStore с улучшенным дизайном

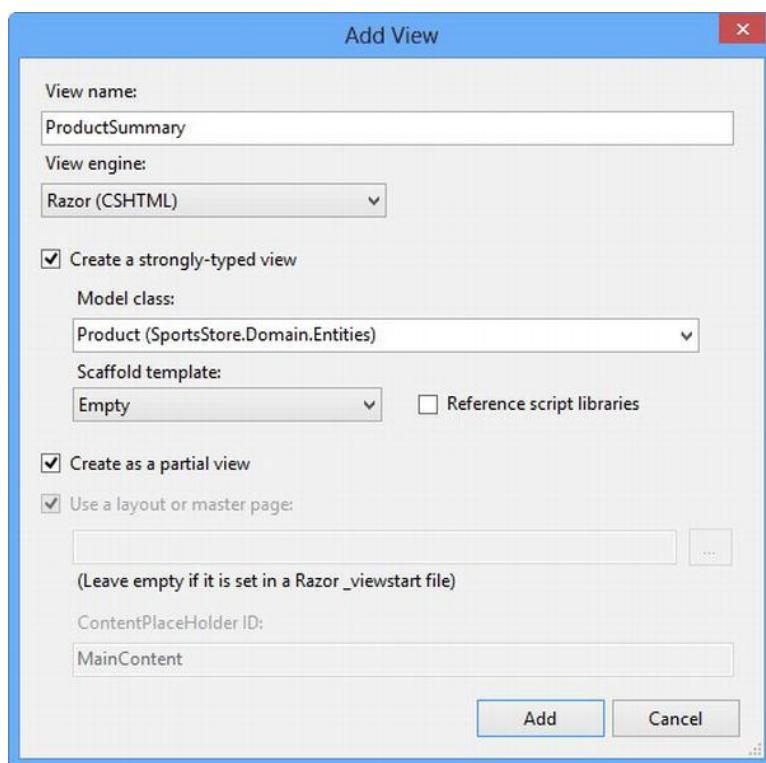


## Создаем частичное представление

Последний прием, который мы изучим этой главе, - это реорганизация приложения с целью упрощения представления `List.cshtml`. Мы создадим частичное представление, которое представляет собой фрагмент содержимого, который внедряется в другое представление. Частичные представления находятся в отдельных файлах и могут использоваться несколькими представлениями, что может помочь сократить дублирование, особенно если вам нужно визуализировать одни и те же данные в приложении несколько раз.

Чтобы добавить частичное представление, кликните правой кнопкой мыши папку `/Views/Shared` в проекте `SportsStore.WebUI` и выберите `Add - View` из контекстного меню. Назовите представление `ProductSummary`. Мы хотим отобразить информацию о товаре, поэтому выберите класс `Product` из выпадающего меню класса `Model` или введите указанное имя класса. Отметьте флажком опцию `Create as a partial view`, как показано на рисунке 7-18.

**Рисунок 7-18:** Создаем частичное представление



Нажмите кнопку `Add`, и Visual Studio создаст файл частичного представления по адресу `Views/Shared/ProductSummary.cshtml`. Частичное представление очень похоже на обычное, за исключением того, что когда оно визуализируется, оно создает фрагмент HTML, а не полный HTML-документ. Если вы откроете представление `ProductSummary`, то увидите, что оно содержит только директиву представления модели, которая является нашим классом доменной модели `Product`. Примените изменения, показанные в листинге 7-26.

**Листинг 7-26:** Добавляем разметку в частичное представление `ProductSummary`

```
@model SportsStore.Domain.Entities.Product

<div class="item">
    <h3>@Model.Name</h3>
    @Model.Description
    <h4>@Model.Price.ToString("c")</h4>
</div>
```

Теперь нам необходимо обновить Views/Products/List.cshtml, чтобы он начал использовать частичное представление. Вы можете увидеть изменения в листинге 7-27.

**Листинг 7-27:** Используем частичное представление в List.cshtml

```
@model SportsStore.WebUI.Models.ProductsListViewModel

{@{
    ViewBag.Title = "Products";
}

@foreach (var p in Model.Products)
{
    Html.RenderPartial("ProductSummary", p);
}

<div class="pager">
    @Html.PageLinks(Model.PagingInfo, x => Url.Action("List", new { page = x }))
</div>
```

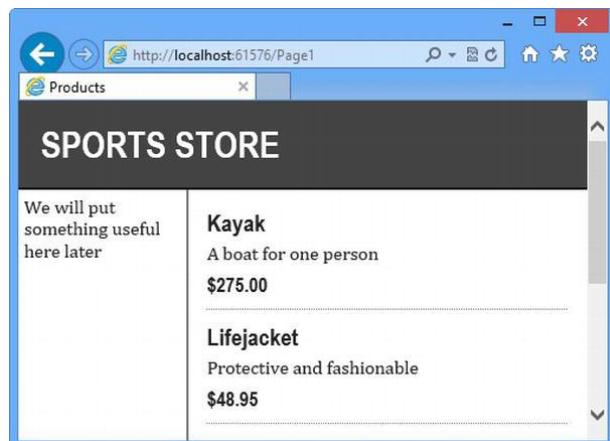
Мы взяли разметку, которая раньше была в цикле `foreach` в представлении List.cshtml, и перенесли ее в новое частичное представление. Мы вызываем частичное представление, используя вспомогательный метод `Html.RenderPartial`. Параметры – это имя представления и объект модели представления.

*Совет*

*Метод `RenderPartial` не возвращает разметку HTML, как большинство других вспомогательных методов. Вместо этого он записывает содержимое непосредственно в поток ответа, и именно поэтому мы должны вызывать его как полную строку C#, используя точку с запятой. Это более эффективно, чем помещение в буфер созданной частичным представлением разметки HTML, потому что потом она все равно будет записана в поток ответа. Если вы предпочитаете более последовательный синтаксис, используйте метод `Html.Partial`, который делает то же, что и метод `RenderPartial`, но возвращает фрагмент HTML и может быть вызван как `@Html.Partial("ProductSummary", p)`.*

Использовать частичное представление – это хороший прием. Он не меняет внешний вид приложения. Если вы запустите его, вы увидите, что дисплей остался тем же, как показано на рисунке 7-19.

**Рисунок 7-19:** Применяем частичное представление



# Резюме

В этой главе мы создали большую часть базовой инфраструктуры для приложения SportsStore. На данный момент в нем еще нет многих функций, которые вы могли бы продемонстрировать клиенту, но уже имеется начало доменной модели с хранилищем товаров, которое поддерживается SQL Server и Entity Framework. У нас есть один контроллер, `ProductController`, который может создавать списки товаров, разбитые на страницы, настроенный DI и чистая и понятная схема URL.

Если вам показалось, что в этой главе было слишком много настроек и мало прогресса, то следующая глава ее уравновесит. Теперь, когда мы разобрались с базовыми элементами, мы можем двигаться вперед и добавить все функции для работы клиента с приложением: навигацию по категориям, корзину и процесс оплаты.

# SportsStore: навигация

В предыдущей главе мы создали основу инфраструктуры приложения SportsStore. Теперь мы будем использовать ее для добавления ключевых функций в приложение, и вы начнете видеть, как окупается тот труд и время, которое мы затратили на создание основы. Мы сможем легко и просто добавлять важные для работы с приложением функции. В процессе вы познакомитесь с некоторыми дополнительными возможностями, которые предоставляет MVC Framework.

## Добавление элементов навигации

Приложение SportsStore будет намного более удобным, если мы позволим пользователям просматривать каталог по категориям. Мы выполним эту работу в три этапа:

- Расширим модель действия `List` в классе `ProductController` так, чтобы она могла фильтровать объекты `Product` в хранилище.
- Пересмотрим и улучшим нашу схему URL и исправим стратегию изменения маршрута.
- Создадим список категорий, который будет размещен в боковой панели сайта, подсветку текущей категории и ссылки на другие категории.

### Фильтрация списка товаров

Мы начнем с расширения класса модели представления, `ProductsListViewModel`, который мы добавили в проект `SportsStore.WebUI` в предыдущей главе. Мы должны обеспечить связь выбранной на данный момент категории с представлением, чтобы визуализировать боковую панель, так что начнем с этого. Изменения показаны в листинге 8-1.

#### Листинг 8-1: Расширяем класс `ProductsListViewModel`

```
using System.Collections.Generic;
using SportsStore.Domain.Entities;
namespace SportsStore.WebUI.Models
{
    public class ProductsListViewModel
    {
        public IEnumerable<Product> Products { get; set; }
        public PagingInfo PagingInfo { get; set; }
        public string CurrentCategory { get; set; }
    }
}
```

Мы добавили новое свойство под названием `CurrentCategory`. Далее мы обновим класс `ProductController`, чтобы метод действия `List` отфильтровывал объекты `Product` по категориям и использовал новое свойство, которое мы добавили к модели представления, чтобы указывать выбранную категорию. Изменения показаны в листинге 8-2.

#### Листинг 8-2: Добавляем поддержку категорий в метод действия `List`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using SportsStore.WebUI.Models;
namespace SportsStore.WebUI.Controllers
```

```

{
    public class ProductController : Controller
    {
        private IProductRepository repository;
        public int PageSize = 4;
        public ProductController(IProductRepository productRepository)
        {
            this.repository = productRepository;
        }
        public ViewResult List(string category, int page = 1)
        {
            ProductsListViewModel viewModel = new ProductsListViewModel
            {
                Products = repository.Products
                    .Where(p => category == null || p.Category == category)
                    .OrderBy(p => p.ProductID)
                    .Skip((page - 1) * PageSize)
                    .Take(PageSize),
                PagingInfo = new PagingInfo
                {
                    CurrentPage = page,
                    ItemsPerPage = PageSize,
                    TotalItems = repository.Products.Count()
                },
                CurrentCategory = category
            };
            return View(viewModel);
        }
    }
}

```

Мы сделали три изменения в методе действия. Во-первых, мы добавили новый параметр под названием `category`. Этот параметр используется вторым изменением, которое представляет собой расширение запроса LINQ: теперь если `category` не содержит `null`, будут выбраны только те объекты `Product`, которые соответствуют свойству `Category`. Последнее изменение заключается в том, что мы установили значение свойства `CurrentCategory`, добавленного в класс `ProductsListViewModel`. Однако, эти изменения означают, что значение `PagingInfo.TotalItems` рассчитываются неправильно, что мы скоро исправим.

## Модульный тест: обновление существующих модульных тестов

Мы изменили сигнатуру метода действия `List`, из-за чего некоторые из наших существующих модульных тестов не будут скомпилированы. Чтобы решить эту проблему, передайте `null` в качестве первого параметра в метод `List` в те модульные тесты, которые работают с контроллером. Например, в teste `Can_Paginate` раздел действия станет таким:

```

...
[TestMethod]
public void Can_Paginate()
{
    // Arrange
    Mock<IProductRepository> mock = new Mock<IProductRepository>();
    mock.Setup(m => m.Products).Returns(new Product[] {
        new Product {ProductID = 1, Name = "P1"},
        new Product {ProductID = 2, Name = "P2"},
        new Product {ProductID = 3, Name = "P3"},
        new Product {ProductID = 4, Name = "P4"},
        new Product {ProductID = 5, Name = "P5"}
    }.AsQueryable());
    // create a controller and make the page size 3 items
    ProductController controller = new ProductController(mock.Object);
}

```

```

controller.PageSize = 3;
// Act
ProductsListViewModel result
    = (ProductsListViewModel)controller.List(null, 2).Model;
// Assert
Product[] prodArray = result.Products.ToArray();
Assert.IsTrue(prodArray.Length == 2);
Assert.AreEqual(prodArray[0].Name, "P4");
Assert.AreEqual(prodArray[1].Name, "P5");
}
...

```

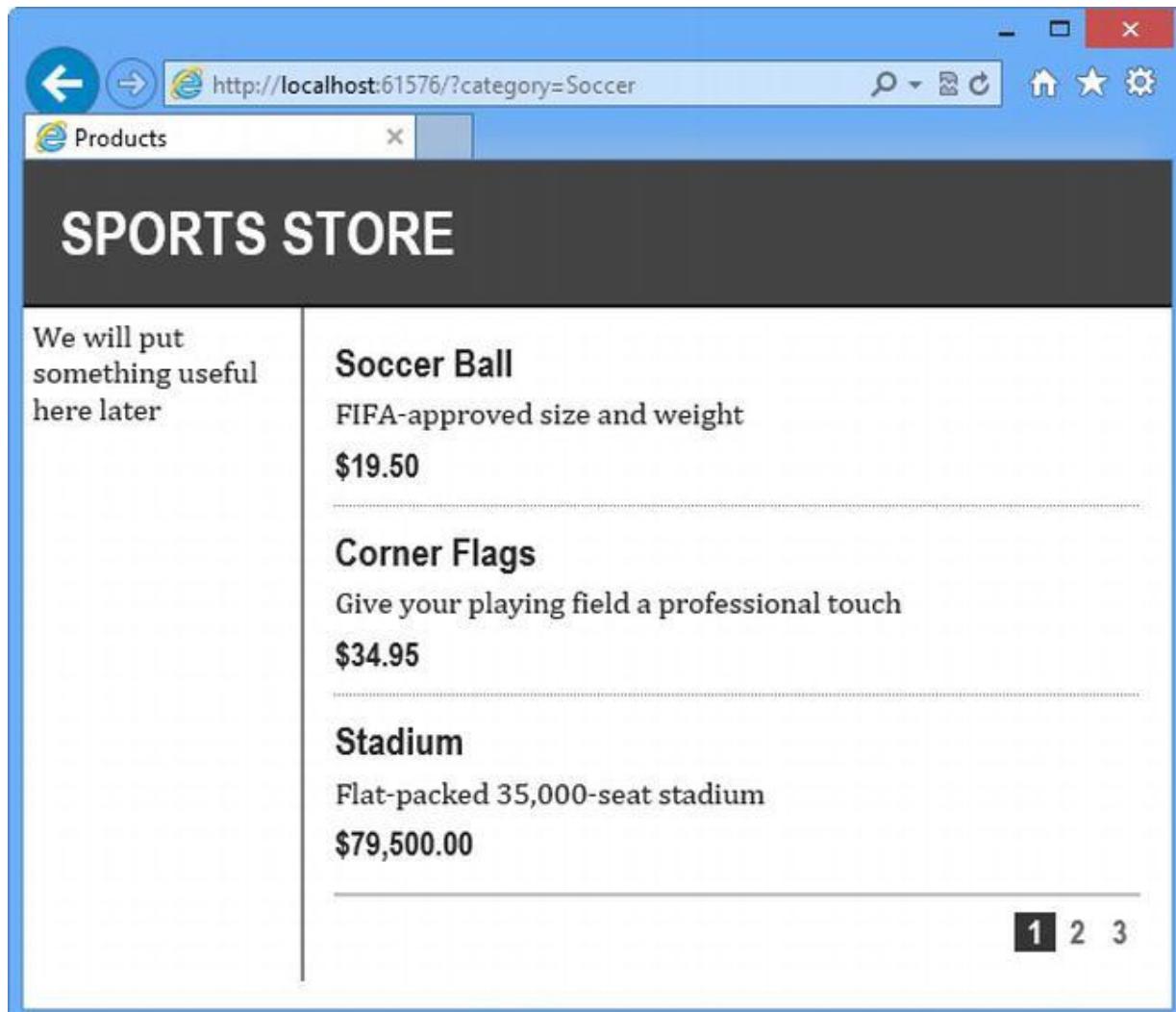
Используя null, мы получаем все объекты Product, которые контроллер получает из хранилища, что полностью повторяет ситуацию, которая была раньше, пока мы не добавили новый параметр.

Даже с этими небольшими изменениями мы можем увидеть эффект фильтрации. Предположим, что вы запускаете приложение и выбираете категорию с помощью строки запроса, например:

`http://localhost:61576/?category=Soccer`

Вы увидите только товары в категории Soccer, как показано на рисунке 8-1.

**Рисунок 8-1:** Использование строки запроса для фильтрации по категориям



## Модульный тест: фильтрация категорий

Нам нужно тщательно протестировать функцию фильтрации по категориям, чтобы гарантировать, что фильтрация проводится корректно, и мы получаем только продукты из указанной категории. Вот тест:

```
...
[TestMethod]
public void Can_Filter_Products()
{
    // Arrange
    // - create the mock repository
    Mock<IProductRepository> mock = new Mock<IProductRepository>();
    mock.Setup(m => m.Products).Returns(new Product[] {
        new Product {ProductID = 1, Name = "P1", Category = "Cat1"},
        new Product {ProductID = 2, Name = "P2", Category = "Cat2"},
        new Product {ProductID = 3, Name = "P3", Category = "Cat1"},
        new Product {ProductID = 4, Name = "P4", Category = "Cat2"},
        new Product {ProductID = 5, Name = "P5", Category = "Cat3"}
    }.AsQueryable());

    // Arrange - create a controller and make the page size 3 items
    ProductController controller = new ProductController(mock.Object);
    controller.PageSize = 3;

    // Action
    Product[] result = ((ProductsListViewModel)controller.List("Cat2", 1).Model)
        .Products.ToArray();

    // Assert
    Assert.AreEqual(result.Length, 2);
    Assert.IsTrue(result[0].Name == "P2" && result[0].Category == "Cat2");
    Assert.IsTrue(result[1].Name == "P4" && result[1].Category == "Cat2");
}
...
}
```

Этот тест создает имитированное хранилище, содержащее объекты `Product`, которые принадлежат к различным категориям. С помощью метода `Action` запрашивается одна определенная категория, и, мы проверяем результаты, чтобы убедиться, что получаем правильные объекты в правильном порядке.

## Уточняем схему URL

Никому не нужны страшные URL вроде `/?category=Soccer`. Чтобы это исправить, мы вернемся к схеме маршрутизации и изменим ее таким образом, чтобы она лучше подходила нам (и нашим пользователям). Для реализации нашей новой схемы, измените метод `RegisterRoutes` в файле `App_Start/RouteConfig.cs` так, чтобы он соответствовал листингу 8-3, заменяя содержимое метода, который мы использовали в предыдущей главе.

### Листинг 8-3: Новая схема URL

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
namespace SportsStore.WebUI
{
    public class RouteConfig
    {
```

```

public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapRoute(null,
        "",
        new
        {
            controller = "Product",
            action = "List",
            category = (string)null,
            page = 1
        }
    );

    routes.MapRoute(null,
        "Page{page}",
        new { controller = "Product", action = "List", category = (string)null },
        new { page = @"\d+" }
    );

    routes.MapRoute(null,
        "{category}",
        new { controller = "Product", action = "List", page = 1 }
    );

    routes.MapRoute(null,
        "{category}/Page{page}",
        new { controller = "Product", action = "List" },
        new { page = @"\d+" }
    );
}

routes.MapRoute(null, "{controller}/{action}");
}
}
}

```

#### *Внимание*

*Важно добавлять новые роуты из листинга 8-3 по очереди, как они показаны в листинге. Роуты применяются в том порядке, в котором они определены, и если вы его измените, то получите другой результат.*

Таблица 8-1 описывает схему URL, которую представляют эти роуты. Мы расскажем о системе маршрутизации подробно в главе 13.

**Таблица 8-1:** Информация о роутах

URL	Результат
/	Выводит список товаров из всех категорий для первой страницы.
/Page2	Выводит список товаров из всех категорий для указанной страницы (в данном случае страницы 2).
/Soccer	Показывает первую страницу товаров из определенной категории (в данном случае категории Soccer).
/Soccer/Page2	Показывает указанную страницу (в данном случае 2) товаров из указанной категории (в данном случае Soccer).
/Anything/Else	Вызывает метод действия Else контроллера Anything.

Система маршрутизации ASP.NET используется MVC для обработки *входящих* запросов от пользователей, но она также запрашивает *исходящие* URL, которые соответствуют нашей схеме URL, и которые мы можем встроить в веб-страницы. Таким образом гарантируется то, что все URL в приложении последовательны.

## Заметка

Мы покажем, как создавать модульные тесты для конфигурации маршрутизации в главе 13.

Метод `Url.Action` является наиболее удобным способом генерации исходящих ссылок. В предыдущей главе мы использовали этот вспомогательный метод в представлении `List.cshtml`, чтобы отображать ссылки на страницы. Теперь, когда мы добавили поддержку фильтрации по категориям, мы должны вернуться к нему и передать эту информацию, как показано в листинге 8-4.

### Листинг 8-4: Добавляем информацию о категории к ссылкам на страницы

```
@model SportsStore.WebUI.Models.ProductsListViewModel

@{
    ViewBag.Title = "Products";
}

@foreach (var p in Model.Products)
{
    Html.RenderPartial("ProductSummary", p);
}

<div class="pager">
    @Html.PageLinks(Model.PagingInfo, x => Url.Action("List",
        new { page = x, category = Model.CurrentCategory }))
</div>
```

До этого изменения ссылки на страницы выглядели так:

```
http://<myserver>:<port>/Page2
```

Если пользователь перейдет по такой ссылке, фильтр по категории будет потерян, и он попадет на страницу, содержащую товары из всех категорий. Добавляя текущую категорию, которую мы получаем из модели представления, мы генерируем такие URL:

```
http://<myserver>:<port>/Chess/Page2
```

Когда пользователь переходит по такой ссылке, текущая категория будут передана в метод действия `List`, и фильтрация будет сохранена. После внесения этих изменений, вы можете перейти по таким ссылкам, как `/Chess` или `/Soccer`, и увидите, что ссылка внизу страницы включает в себя правильную категорию.

## Создаем меню навигации по категориям

Мы должны предоставить пользователям возможность выбора категории. Это означает, что мы должны создать список доступных категорий, в котором будет выделяться выбранная категория, если такая имеется. В процессе работы над приложением мы будем использовать этот список в разных контроллерах, поэтому он должен быть реализован отдельно и предоставлять возможность многократного использования.

В ASP.NET MVC Framework есть концепция дочерних действий, которые идеально подходят для создания таких элементов, как элемент управления навигацией многократного использования.

Дочернее действие полагается на вспомогательный метод HTML под названием `RenderAction`, который позволяет включить вывод из произвольного метода действия в текущее представление. В этом случае мы можем создать новый контроллер (назовем его `NavController`) с методом действия (в данном случае `Menu`), который визуализирует меню навигации и внедряет вывод из данного метода в макет.

Такой подход дает нам реальный контроллер, который может содержать любую необходимую нам логику приложения, и который может быть протестирован, как и любой другой контроллер. Это действительно хороший способ создания небольших сегментов приложения, при котором сохраняется общий подход MVC Framework.

## Создаем контроллер навигации

Щелкните правой кнопкой мыши папку `Controllers` в проекте `SportsStore.WebUI` и выберите пункт `Add Controller` из контекстного меню. Назовите новый контроллер `NavController`, выберите опцию `Empty MVC controller` из меню `Template` и нажмите кнопку `Add to create the class`.

Удалите метод `Index`, который Visual Studio создает по умолчанию, и добавьте метод действия `Menu`, показанный в листинге 8-5.

### Листинг 8-5: Метод действия Menu

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
namespace SportsStore.WebUI.Controllers
{
    public class NavController : Controller
    {
        public string Menu()
        {
            return "Hello from NavController";
        }
    }
}
```

Этот метод возвращает статическую строку сообщения, но, пока мы интегрируем дочернее действие в приложение, этого для нас достаточно. Мы хотим, чтобы список категорий появлялся на всех страницах, так что мы собираемся визуализировать дочернее действие в макете, а не в определенном представлении. Отредактируйте файл `Views/Shared/_Layout.cshtml` так, чтобы он вызывал вспомогательный метод `RenderAction`, как показано в листинге 8-6.

### Листинг 8-6: Добавляем вызов к `RenderAction` в макет Razor

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    <link href="~/Content/Site.css" type="text/css" rel="stylesheet" />
</head>
<body>
    <div id="header">
        <div class="title">SPORTS STORE</div>
    </div>
    <div id="categories">
```

```

@{ Html.RenderAction("Menu", "Nav"); }
</div>
<div id="content">
    @RenderBody()
</div>
</body>
</html>

```

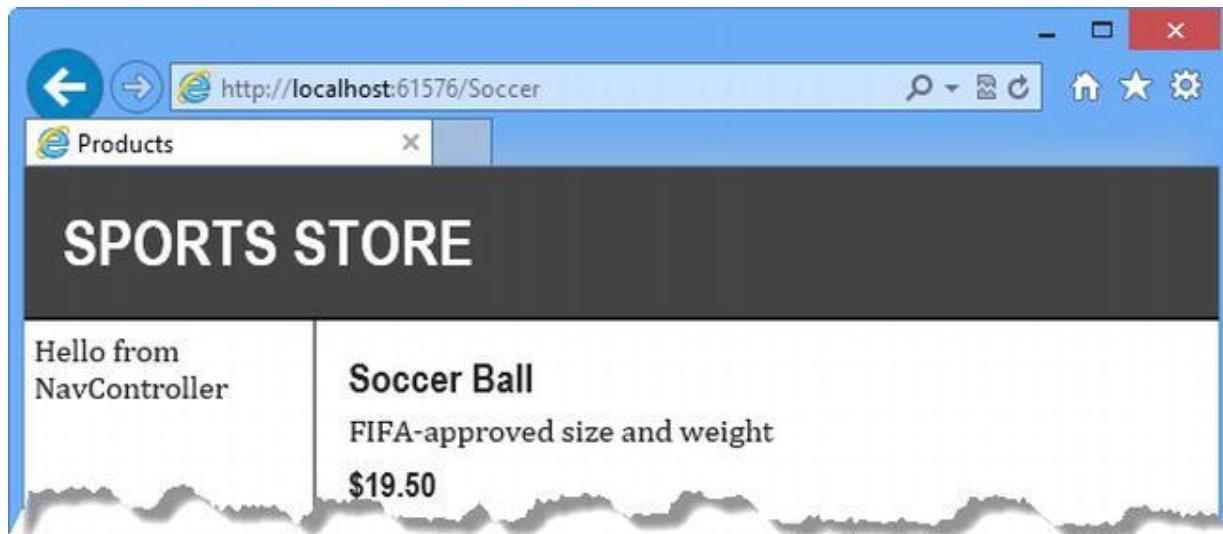
Мы удалили замещающий текст, который добавили в главе 7, и заменили его на вызов метода `RenderAction`. Параметрами этого метода являются метод действия, который мы хотим вызывать (`Menu`), и контроллер, который мы хотим использовать (`Nav`).

### Примечание

Метод `RenderAction` записывает свое содержание непосредственно в поток ответа, как и метод `RenderPartial`, о котором мы упоминали в главе 5. Это означает, что метод возвращает `void`, и поэтому его нельзя использовать с регулярным тегом Razor `@`. Вместо этого мы должны заключить вызов метода в блок кода Razor (и не забудьте поставить точку с запятой в конце оператора). Если вам не нравится синтаксис блока кода, можно использовать метод `Action` в качестве альтернативы.

Если вы запустите приложение, то увидите, что вывод метода действия `Menu` включен в каждую страницу, как показано на рисунке 8-2.

**Рисунок 8-2:** Отображение результата метода действия `Menu`



### Создаем списки категорий

Теперь мы можем вернуться к контроллеру и создать реальный набор категорий. Мы не хотим генерировать категории URL в контроллере. Для этого мы собираемся использовать вспомогательный метод в представлении. В методе действия `Menu` нужно только создать список категорий, что мы сделали в листинге 8-7.

**Листинг 8-7:** Реализация метода `Menu`

```

using SportsStore.Domain.Abstract;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

```

```

namespace SportsStore.WebUI.Controllers
{
    public class NavController : Controller
    {
        private IProductRepository repository;

        public NavController(IProductRepository repo)
        {
            repository = repo;
        }

        public PartialViewResult Menu()
        {
            IEnumerable<string> categories = repository.Products
                .Select(x => x.Category)
                .Distinct()
                .OrderBy(x => x);
            return PartialView(categories);
        }
    }
}

```

Сначала мы добавляем конструктор, который принимает реализацию `IProductRepository` как аргумент - после создания экземпляра контроллера ее предоставит Ninject, используя привязки, которые мы создали в предыдущей главе.

Далее мы изменяем метод действия `Menu`, который теперь использует запрос LINQ, чтобы получить список категорий из хранилища и передать их в представление. Обратите внимание, что, так как в этом контроллере мы работаем с частичным представлением, здесь мы вызываем метод `PartialView`, и что результатом является объект `PartialViewResult`.

## Модульный тест: создание списка категорий

Протестировать нашу способность создавать список категорий относительно просто. Наша цель - создать список, который отсортирован в алфавитном порядке и не содержит дубликатов. Самый простой способ это сделать - предоставить неотсортированные тестовые данные с дублирующими категориями, передать их в `NavController` и задать утверждение, что данные будут правильно обработаны. Вот модульный тест, который мы использовали:

```

...
[TestMethod]
public void Can_Create_Categories()
{
    // Arrange
    // - create the mock repository
    Mock<IProductRepository> mock = new Mock<IProductRepository>();
    mock.Setup(m => m.Products).Returns(new Product[] {
        new Product {ProductID = 1, Name = "P1", Category = "Apples"},
        new Product {ProductID = 2, Name = "P2", Category = "Apples"},
        new Product {ProductID = 3, Name = "P3", Category = "Plums"},
        new Product {ProductID = 4, Name = "P4", Category = "Oranges"},
    }.AsQueryable());

    // Arrange - create the controller
    NavController target = new NavController(mock.Object);

    // Act = get the set of categories
    string[] results = ((IEnumerable<string>)target.Menu().Model).ToArray();

    // Assert
    Assert.AreEqual(results.Length, 3);
}

```

```

        Assert.AreEqual(results[0], "Apples");
        Assert.AreEqual(results[1], "Oranges");
        Assert.AreEqual(results[2], "Plums");
    }
...

```

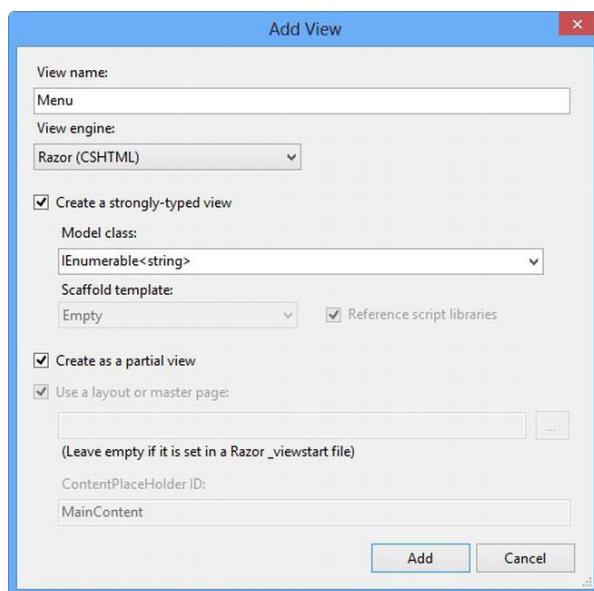
Мы создали имитированную реализацию хранилища, которая содержит повторяющиеся и неотсортированные категории. Наше утверждение заключается в том, что все повторяющиеся строки будут удалены и данные будут отсортированы в алфавитном порядке.

## Создаем частичное представление

Так как список категорий является всего лишь частью страницы, имеет смысл создать частичное представление для метода действия Menu. Кликните правой кнопкой мыши метод Menu в классе NavController и выберите Add View из контекстного меню.

Оставьте представлению имя Menu, отметьте флажком опцию Create a strongly typed view, и введите IEnumerable<string> как тип класса модели, как показано на рисунке 8-3.

**Рисунок 8-3 : Создаем частичное представление Menu**



Отметьте флажком опцию Create as a partial view и нажмите кнопку Add, чтобы создать представление. Измените содержание представления так, чтобы оно соответствовало листингу 8-8.

### Листинг 8-8: Частичное представление Menu

```

@model IEnumerable<string>

@Html.ActionLink("Home", "List", "Product")

@foreach (var link in Model)
{
    @Html.RouteLink(link, new
    {
        controller = "Product",
        action = "List",
        category = link,
        page = 1
    })
}

```

Мы добавили ссылку под названием Home, которая будет отображаться в верхней части списка категорий и приведет пользователя на первую страницу со списком всех товаров, без фильтра по категории. Мы сделали это с помощью вспомогательного метода ActionLink, который генерирует якорный HTML-элемент с помощью информации о маршрутизации, которую мы настроили ранее.

Затем мы перечислили имена категорий и создали ссылки на каждую из них с помощью метода RouteLink. Он похож на ActionLink, но позволяет нам поставлять набор пар имя/значение, которые учитываются при генерации URL на основе конфигурации маршрутизации. Не беспокойтесь, если вы еще ничего не знаете о маршрутизации – мы подробно объясним все в главе 13.

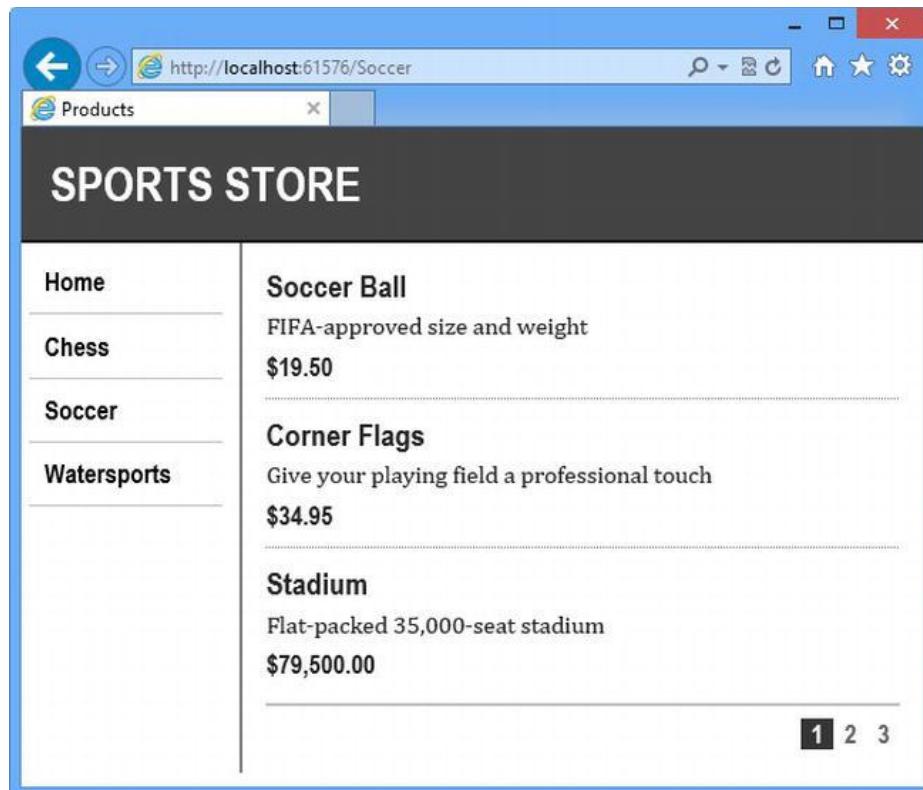
Генерируемые ссылки будут выглядеть не очень симпатично с настройками по умолчанию, поэтому мы определили код CSS, который улучшит их внешний вид. Добавьте стили, показанные в листинге 8-9, в конец файла Content/Site.css в проекте SportsStore.WebUI.

#### Листинг 8-9: CSS для ссылок на категории

```
...
DIV#categories A
{
    font: bold 1.1em "Arial Narrow", "Franklin Gothic Medium", Arial; display: block;
    text-decoration: none; padding: .6em; color: Black;
    border-bottom: 1px solid silver;
}
DIV#categories A.selected { background-color: #666; color: White; }
DIV#categories A:hover { background-color: #CCC; }
DIV#categories A.selected:hover { background-color: #666; }
...
```

Если вы запустите приложение, то увидите ссылки на категории, как показано на рисунке 8-4. Если вы кликните по категории, список элементов обновится и будет отображать только элементы из выбранной категории.

Рисунок 8-4: Ссылки на категории



## Подсветка текущей категории

Сейчас мы не подсказываем пользователям, какую категорию они просматривают. Хотя пользователь может понять это по элементам в списке, мы все же предпочитаем обеспечить надежный визуальный индикатор. Для этого мы могли бы создать модель представления, которая содержит список категорий и выбранную категорию; в самом деле, именно это мы бы обычно и сделали. Но для разнообразия мы будем использовать ViewBag, о которой говорилось в главе 2. Этот объект позволяет передавать данные из контроллера в представление, не используя модель представления. Листинг 8-10 показывает изменения в методе действия Menu контроллера Nav.

**Листинг 8-10:** Использование ViewBag

```
using SportsStore.Domain.Abstract;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
namespace SportsStore.WebUI.Controllers
{
    public class NavController : Controller
    {
        private IProductRepository repository;
        public NavController(IProductRepository repo)
        {
            repository = repo;
        }
        public PartialViewResult Menu(string category = null)
        {
            ViewBag.SelectedCategory = category;
            IEnumerable<string> categories = repository.Products
                .Select(x => x.Category)
                .Distinct()
                .OrderBy(x => x);
            return PartialView(categories);
        }
    }
}
```

Мы добавили в метод действия Menu параметр под названием category. Значение этого параметра будет предоставлено автоматически конфигурацией маршрутизации. В теле метода мы динамически создали свойство SelectedCategory в объекте ViewBag и приравняли его значение к значению параметра category. Как мы уже объясняли в главе 2, ViewBag является динамическим объектом, и мы создаем новые свойства, просто устанавливая для них значения.

## Модульный тест: Указание выбранной категории

Чтобы проверить, что метод действия Menu правильно добавляет информацию о выбранной категории, проверим в модульном teste значение свойства ViewBag, которое доступно через класс ViewResult. Вот этот тест:

```
[TestMethod]
public void Indicates_Selected_Category()
{
    // Arrange
    // - create the mock repository
    Mock<IProductRepository> mock = new Mock<IProductRepository>();
    mock.Setup(m => m.Products).Returns(new Product[] {
        new Product {ProductID = 1, Name = "P1", Category = "Apples"},
```

```

new Product {ProductID = 4, Name = "P2", Category = "Oranges"} ,
}.AsQueryable();

// Arrange - create the controller
NavController target = new NavController(mock.Object);

// Arrange - define the category to selected
string categoryToSelect = "Apples";

// Action
string result = target.Menu(categoryToSelect).ViewBag.SelectedCategory;

// Assert
Assert.AreEqual(categoryToSelect, result);
}

```

Обратите внимание, что мы не должны приводить значение свойства из ViewBag. Это одно из преимуществ использования объекта ViewBag перед ViewData.

Теперь, когда мы предоставляем информацию о выбранной категории, можно обновить представление и добавить класс CSS к якорному HTML-элементу, который представляет выбранную категорию. Листинг 8-11 показывает изменения в частичном представлении Menu.cshtml.

### Листинг 8-11: Подсветка выбранной категории

```

@model IEnumerable<string>

@Html.ActionLink("Home", "List", "Product")

@foreach (var link in Model) {
    @Html.RouteLink(link, new {
        controller = "Product",
        action = "List",
        category = link,
        page = 1
    },
    new {
        @class = link == ViewBag.SelectedCategory ? "selected" : null
    })
}

```

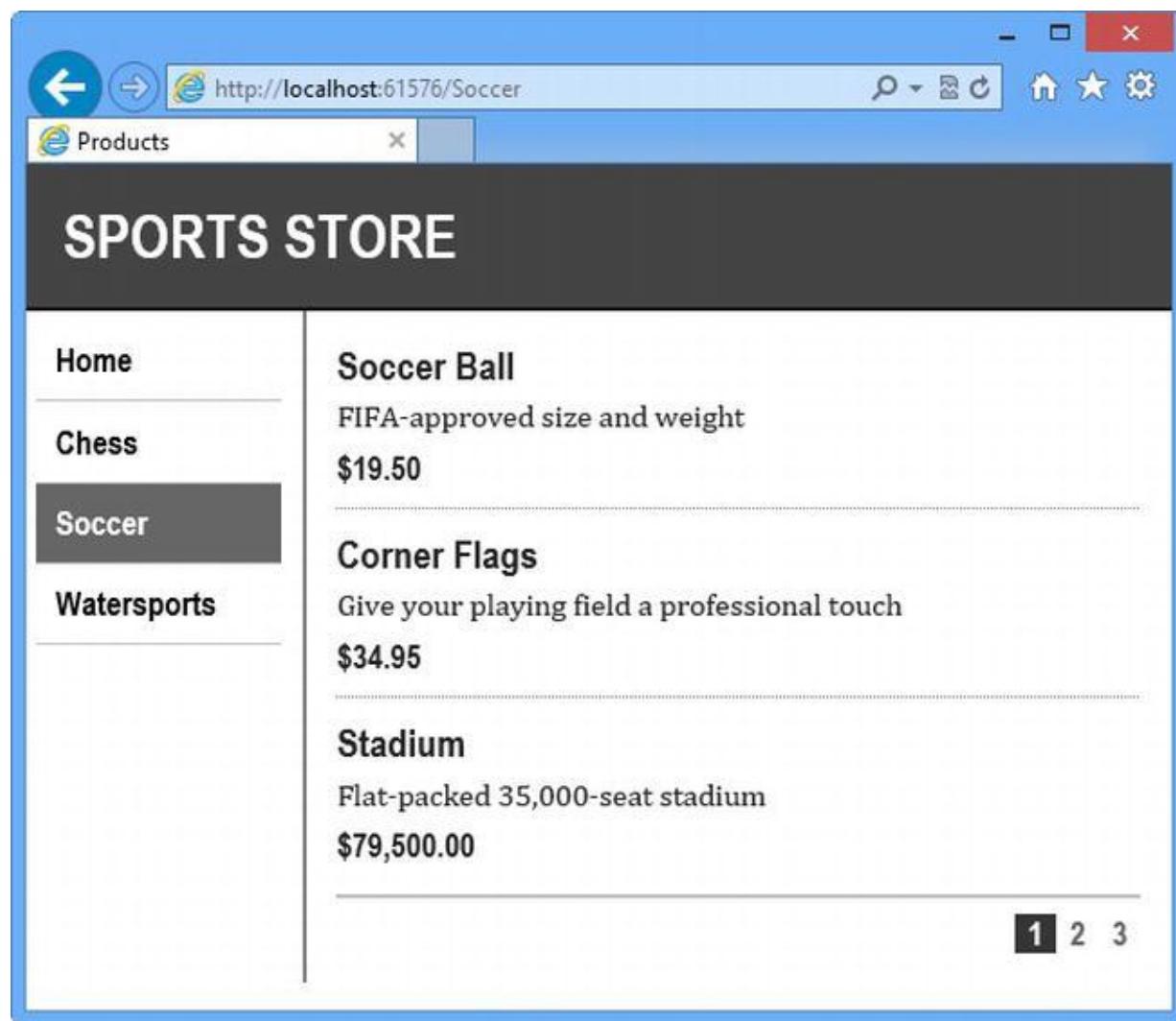
Мы воспользовались перегруженной версией метода RouteLink, что позволяет нам предоставить объект, свойства которого будут добавлены в якорный HTML-элемент как атрибуты. В этом случае ссылке, которая представляет выбранную категорию, присваивается CSS-класс selected.

#### *Примечание*

*Обратите внимание, что мы использовали @class в анонимном объекте, который мы передали как новый параметр в вспомогательный метод RouteLink. Это не тег Razor. Мы используем стандартную функцию языка C#, чтобы избежать конфликта между ключевым словом HTML class (используется для присвоения стиля CSS к элементу) и того же слова C# (используется для обозначения класса .NET). Символ @ позволяет нам использовать зарезервированные ключевые слова C#, не запутывая компилятор. Если мы просто вызовем параметр class (без @), компилятор будет считать, что мы определяем новый тип C#. Когда мы будем использовать символ @, компилятор поймет, что мы хотим создать параметр в анонимном типе под названием class, и мы получим нужный нам результат.*

При запуске приложения будет виден эффект подсветки категории, которую вы также можете видеть на рисунке 8-5.

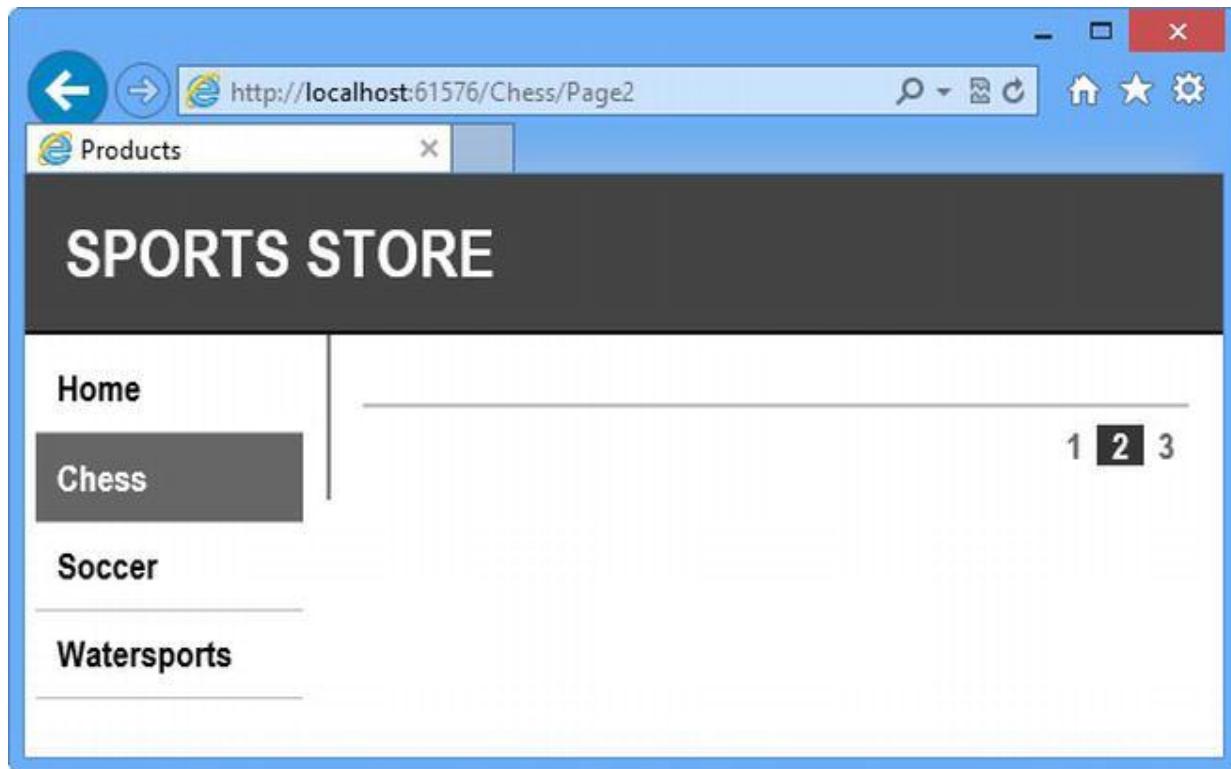
**Рисунок 8-5:** Подсветка выбранной категории



## Корректируем количество страниц

Нам нужно исправить ссылки на страницы, чтобы они работали правильно, когда выбрана категория. На данный момент количество ссылок на страницы определяется общим количеством товаров в хранилище, а не количеством товаров в выбранной категории. Это означает, что клиент может кликнуть по ссылке на страницу 2 в категории `Chess` и попадет на пустую страницу, потому что для ее заполнения недостаточно товаров. Вы можете увидеть, как это выглядит, на рисунке 8-6.

**Рисунок 8-6:** Отображение неправильных ссылок на страницы, когда выбрана категория



Мы можем исправить это, обновив метод действия `List` в `ProductController` так, чтобы к информации о нумерации страниц были добавлены сведения о категории. Необходимые изменения показаны в листинге 8-12.

**Листинг 8-12:** Объединяем данные о нумерации страниц и категории

```
public ViewResult List(string category, int page = 1)
{
    ProductsListViewModel viewModel = new ProductsListViewModel
    {
        Products = repository.Products
            .Where(p => category == null || p.Category == category)
            .OrderBy(p => p.ProductID)
            .Skip((page - 1) * PageSize)
            .Take(PageSize),
        PagingInfo = new PagingInfo
        {
            CurrentPage = page,
            ItemsPerPage = PageSize,
            TotalItems = category == null ?
                repository.Products.Count() :
                repository.Products.Where(e => e.Category == category).Count()
        },
        CurrentCategory = category
    };
    return View(viewModel);
}
```

Если категория выбрана, мы возвращаем количество товаров в этой категории, если нет, мы возвращаем общее количество товаров.

## Модульный тест: подсчет товаров по категориям

Тест, с помощью которого мы проверим текущий подсчет товаров в различных категориях, очень простой: мы создадим имитированное хранилище, которое содержит известное количество данных в различных категориях, а затем вызовем метод действия `List`, запрашивая каждую категорию по очереди. Вот модульный тест:

```
[TestMethod]
public void Generate_Category_Specific_Product_Count()
{
    // Arrange
    // - create the mock repository
    Mock<IProductRepository> mock = new Mock<IProductRepository>();
    mock.Setup(m => m.Products).Returns(new Product[] {
        new Product {ProductID = 1, Name = "P1", Category = "Cat1"},
        new Product {ProductID = 2, Name = "P2", Category = "Cat2"},
        new Product {ProductID = 3, Name = "P3", Category = "Cat1"},
        new Product {ProductID = 4, Name = "P4", Category = "Cat2"},
        new Product {ProductID = 5, Name = "P5", Category = "Cat3"}
    }.AsQueryable());

    // Arrange - create a controller and make the page size 3 items
    ProductController target = new ProductController(mock.Object);
    target.PageSize = 3;

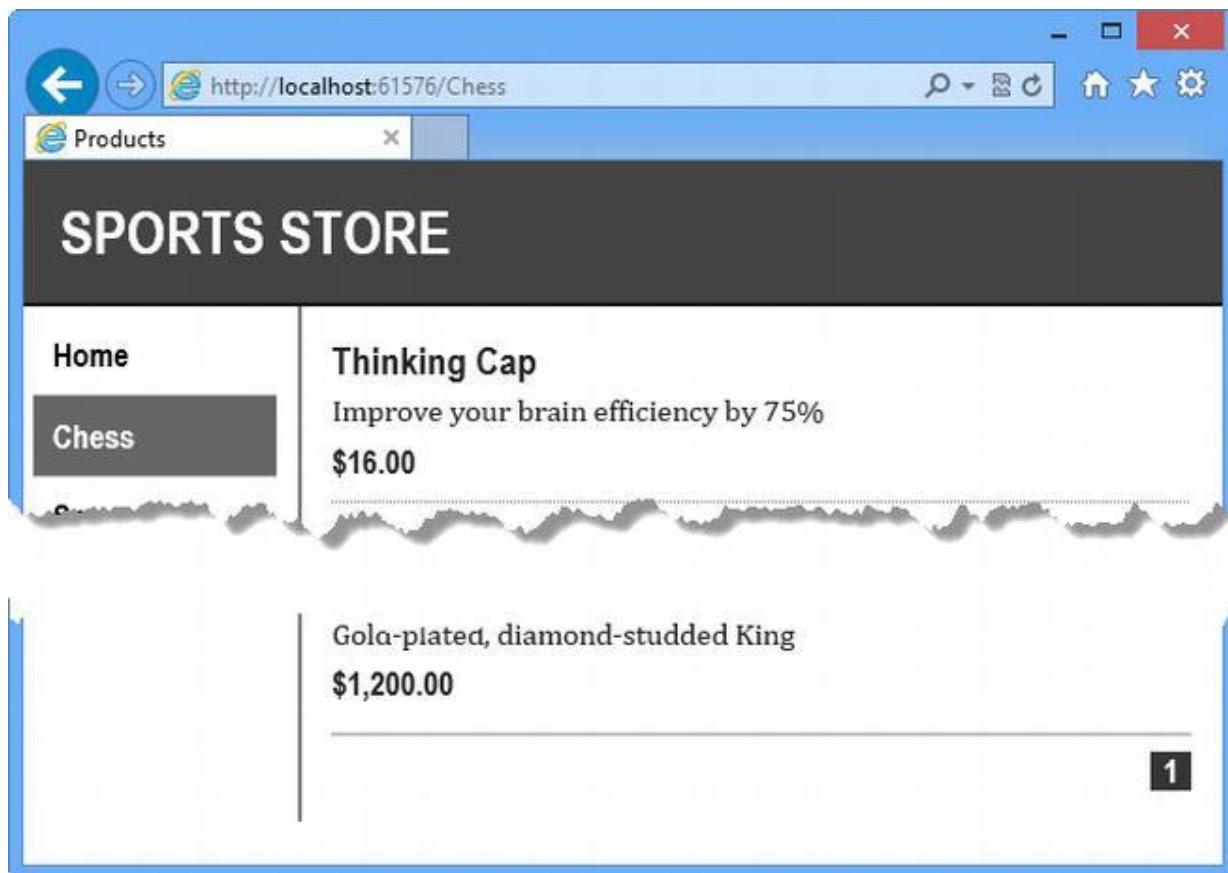
    // Action - test the product counts for different categories
    int res1 = ((ProductsListViewModel)target
        .List("Cat1").Model).PagingInfo.TotalItems;
    int res2 = ((ProductsListViewModel)target
        .List("Cat2").Model).PagingInfo.TotalItems;
    int res3 = ((ProductsListViewModel)target
        .List("Cat3").Model).PagingInfo.TotalItems;
    int resAll = ((ProductsListViewModel)target
        .List(null).Model).PagingInfo.TotalItems;

    // Assert
    Assert.AreEqual(res1, 2);
    Assert.AreEqual(res2, 2);
    Assert.AreEqual(res3, 1);
    Assert.AreEqual(resAll, 5);
}
```

Обратите внимание, что мы также вызываем метод `List`, не указывая категорию, чтобы убедиться, мы получаем правильный подсчет всех товаров.

Теперь, когда мы просматриваем какую-либо категорию, ссылки в нижней части страницы отражают правильное количество товаров в ней, как показано на рисунке 8-7.

Рисунок 8-7: Отображается правильное количество страниц в категории



## Создание корзины покупателя

Наше приложение хорошо развивается, но мы не можем продавать какие-либо товары, пока не реализуем корзину. В этом разделе мы создадим функционал, показанный на рисунке 8-8. Он будет знаком любому человеку, который когда-либо делал покупки в интернете.

Рисунок 8-8: Базовый поток корзины



Кнопка `Add to cart` будет отображаться рядом с каждым из продуктов в нашем каталоге. После нажатия этой кнопки будет отображена информация о товарах, которые клиент уже выбрал, и их общая стоимость. В этот момент пользователь может нажать кнопку `Continue shopping`, чтобы вернуться в каталог товаров, или нажать кнопку `Checkout now`, чтобы выполнить заказ и завершить сессию.

## Определяем сущность корзины

Корзина является частью бизнес-логики нашего приложения, так что имеет смысл представить ее, создав сущность в нашей доменной модели. Добавьте файл класса под названием `Cart` в папку `Entities` проекта `SportsStore.Domain` и определите классы, показанные в листинге 8-13.

### Листинг 8-13: Классы `Cart` и `CartLine`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace SportsStore.Domain.Entities
{
    public class Cart
    {
        private List<CartLine> lineCollection = new List<CartLine>();

        public void AddItem(Product product, int quantity)
        {
            CartLine line = lineCollection
                .Where(p => p.Product.ProductID == product.ProductID)
                .FirstOrDefault();

            if (line == null)
            {
                lineCollection.Add(new CartLine
                {
                    Product = product,
                    Quantity = quantity
                });
            }
            else
            {
                line.Quantity += quantity;
            }
        }

        public void RemoveLine(Product product)
        {
            lineCollection.RemoveAll(l => l.Product.ProductID == product.ProductID);
        }

        public decimal ComputeTotalValue()
        {
            return lineCollection.Sum(e => e.Product.Price * e.Quantity);
        }

        public void Clear()
        {
            lineCollection.Clear();
        }

        public IEnumerable<CartLine> Lines
        {
            get { return lineCollection; }
        }
    }

    public class CartLine
    {
        public Product Product { get; set; }
        public int Quantity { get; set; }
    }
}
```

Класс `Cart` использует `CartLine`, определенный в том же файле, чтобы представлять товар, выбранный покупателем, и количество данного товара. Мы определили методы, которые позволяют добавлять товар в корзину, удалять ранее добавленный товар, рассчитывать общую стоимость товаров в корзине и очистить корзину, удалив все выбранное. Мы также предоставили свойство, которое дает доступ к содержимому корзины с помощью `IEnumerable<CartLine>`. Это все очень простые вещи, которые легко реализовать с помощью C# и немного LINQ.

## Модульный тест: тестирование корзины

Класс `Cart` относительно простой, но у него есть ряд важных линий поведения, и мы должны гарантировать, что они работают должным образом. Плохо функционирующая корзина подорвет все приложение `SportsStore`. Мы разобрали все функции и протестировали их индивидуально. Для этих тестов мы создали новый файл модульных тестов в проекте `SportsStore.UnitTests` под названием `CartTests.cs`. Первая линия поведения относится к добавлению элемента в корзину. Если данный объект `Product` добавляется в корзину в первый раз, то мы хотим, чтобы был добавлен новый объект `CartLine`. Ниже приведен тест и определение класса модульного тестирования:

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using SportsStore.Domain.Entities;
using System.Linq;
namespace SportsStore.UnitTests
{

    [TestClass]
    public class CartTests
    {
        [TestMethod]
        public void Can_Add_New_Lines()
        {
            // Arrange - create some test products
            Product p1 = new Product { ProductID = 1, Name = "P1" };
            Product p2 = new Product { ProductID = 2, Name = "P2" };

            // Arrange - create a new cart
            Cart target = new Cart();
            // Act
            target.AddItem(p1, 1);
            target.AddItem(p2, 1);
            CartLine[] results = target.Lines.ToArray();

            // Assert
            Assert.AreEqual(results.Length, 2);
            Assert.AreEqual(results[0].Product, p1);
            Assert.AreEqual(results[1].Product, p2);
        }
    }
}
```

Однако, если `Product` уже есть в корзине, мы хотим увеличить количество в соответствующем объекте `CartLine` и не создавать новый. Вот тест:

```
[TestMethod]
public void Can_Add_Quantity_For_Existing_Lines()
{
    // Arrange - create some test products
    Product p1 = new Product { ProductID = 1, Name = "P1" };
    Product p2 = new Product { ProductID = 2, Name = "P2" };

    // Arrange - create a new cart
    Cart target = new Cart();
```

```

Cart target = new Cart();
// Act
target.AddItem(p1, 1);
target.AddItem(p2, 1);
target.AddItem(p1, 10);
CartLine[] results = target.Lines.OrderBy(c => c.Product.ProductID).ToArray();

// Assert
Assert.AreEqual(results.Length, 2);
Assert.AreEqual(results[0].Quantity, 11);
Assert.AreEqual(results[1].Quantity, 1);
}

```

Мы также должны убедиться, что пользователи могут передумать и удалить товары из корзины. Эта функция реализуется с помощью метода RemoveLine. Вот тест:

```

[TestMethod]
public void Can_Remove_Line()
{
    // Arrange - create some test products
    Product p1 = new Product { ProductID = 1, Name = "P1" };
    Product p2 = new Product { ProductID = 2, Name = "P2" };
    Product p3 = new Product { ProductID = 3, Name = "P3" };

    // Arrange - create a new cart
    Cart target = new Cart();

    // Arrange - add some products to the cart
    target.AddItem(p1, 1);
    target.AddItem(p2, 3);
    target.AddItem(p3, 5);
    target.AddItem(p2, 1);

    // Act
    target.RemoveLine(p2);

    // Assert
    Assert.AreEqual(target.Lines.Where(c => c.Product == p2).Count(), 0);
    Assert.AreEqual(target.Lines.Count(), 2);
}

```

Следующая линия поведения, которую мы хотим протестировать, - это расчет общей стоимости товаров в корзине. Вот тест:

```

[TestMethod]
public void Calculate_Cart_Total()
{
    // Arrange - create some test products
    Product p1 = new Product { ProductID = 1, Name = "P1", Price = 100M };
    Product p2 = new Product { ProductID = 2, Name = "P2", Price = 50M };

    // Arrange - create a new cart
    Cart target = new Cart();

    // Act
    target.AddItem(p1, 1);
    target.AddItem(p2, 1);
    target.AddItem(p1, 3);
    decimal result = target.ComputeTotalValue();

    // Assert
    Assert.AreEqual(result, 450M);
}

```

Последний тест очень простой. Мы хотим гарантировать, что содержимое корзины удаляется, когда мы ее очищаем. Вот тест:

```
[TestMethod]
public void Can_Clear_Contents()
{
    // Arrange - create some test products
    Product p1 = new Product { ProductID = 1, Name = "P1", Price = 100M };
    Product p2 = new Product { ProductID = 2, Name = "P2", Price = 50M };

    // Arrange - create a new cart
    Cart target = new Cart();

    // Arrange - add some items
    target.AddItem(p1, 1);
    target.AddItem(p2, 1);

    // Act - reset the cart
    target.Clear();

    // Assert
    Assert.AreEqual(target.Lines.Count(), 0);
}
```

Иногда в таких случаях код, необходимый для проверки функциональности какого-либо типа, намного длиннее и сложнее, чем сам тип. Не позволяйте этому оттолкнуть вас от написания модульных тестов. Дефекты в простых классах, особенно таких важных, как `Cart` в нашем приложении, могут иметь серьезные последствия.

## Добавляем кнопку Add to Cart

Чтобы добавить кнопки к спискам товаров, нам нужно изменить частичное представление `Views/Shared/ProductSummary.cshtml`. Изменения показаны в листинге 8-14.

**Листинг 8-14:** Добавляем кнопки в частичное представление `ProductSummary`

```
@model SportsStore.Domain.Entities.Product

<div class="item">
    <h3>@Model.Name</h3>
    @Model.Description
    @using (Html.BeginForm("AddToCart", "Cart")) {
        @Html.HiddenFor(x => x.ProductID)
        @Html.Hidden("returnUrl", Request.Url.PathAndQuery)
        <input type="submit" value="+ Add to cart" />
    }

    <h4>@Model.Price.ToString("c")</h4>
</div>
```

Мы добавили блок Razor, который создает небольшую HTML-форму для каждого товара в списке. Отправка этой формы вызовет метод действия `AddToCart` в контроллере `Cart` (мы скоро реализуем этот метод).

### Примечание

*По умолчанию вспомогательный метод `BeginForm` создает форму, которая использует метод HTTP POST. Вы можете изменить это так, чтобы формы использовали метод GET, но об этом нужно хорошо подумать. Спецификация HTTP*

*требует, чтобы запросы GET были идемпотентными, что означает, что они не должны вызывать изменений, а добавление товара в корзину, безусловно, является изменением. Об этом мы подробнее поговорим в главе 14, в которой и объясним последствия игнорирования идемпотентных запросов GET.*

---

Мы хотим, чтобы стиль этих кнопок соответствовал стилю всего приложения, поэтому добавьте в конец файла Content/Site.css код CSS, показанный в листинге 8-15.

#### Листинг 8-15: Применяем стили к кнопкам

```
FORM { margin: 0; padding: 0; }
DIV.item FORM { float:right; }
DIV.item INPUT {
    color:White; background-color: #333; border: 1px solid black; cursor:pointer;
}
```

#### Создаем несколько HTML-форм на странице

Использование вспомогательного метода `Html.BeginForm` в каждом списке товаров означает, что каждая кнопка `Add to cart` визуализируется в своем отдельном HTML-элементе `form`. Это может вас удивить, если раньше вы работали с ASP.NET Web Forms, где количество форм на странице ограничено одной. В ASP.NET MVC нет лимита на количество форм на странице, у вас их может быть столько, сколько вам нужно.

Нет такого технического требования, согласно которому мы должны создавать форму для каждой кнопки. Однако, так как каждая форма будет отправлять данные к одному и тому же методу контроллера, но с разными значениями параметров, будет лучше и проще поработать с нажатиями кнопок.

#### Реализуем Cart Controller

Нам нужно создать контроллер для обработки нажатий кнопки `Add to cart`. Создайте новый контроллер под названием `CartController` и отредактируйте его содержимое так, чтобы он соответствовал листингу 8-16.

#### Листинг 8-16: Создем CartController

```
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace SportsStore.WebUI.Controllers
{
    public class CartController : Controller
    {
        private IProductRepository repository;

        public CartController(IProductRepository repo)
        {
            repository = repo;
        }

        public RedirectToRouteResult AddToCart(int productId, string returnUrl)
        {
```

```

Product product = repository.Products
    .FirstOrDefault(p => p.ProductID == productId);
if (product != null)
{
    GetCart().AddItem(product, 1);
}
return RedirectToAction("Index", new { returnUrl });
}

public RedirectToRouteResult RemoveFromCart(int productId, string returnUrl)
{
    Product product = repository.Products
        .FirstOrDefault(p => p.ProductID == productId);
    if (product != null)
    {
        GetCart().RemoveLine(product);
    }
    return RedirectToAction("Index", new { returnUrl });
}

private Cart GetCart()
{
    Cart cart = (Cart)Session["Cart"];
    if (cart == null)
    {
        cart = new Cart();
        Session["Cart"] = cart;
    }
    return cart;
}
}
}

```

По поводу этого контроллера есть несколько замечаний. Первое касается того, что мы используем состояние сессии ASP.NET для сохранения и извлечения объектов Cart. Это задача метода GetCart. В ASP.NET есть объект Session, который использует cookie или перезапись URL для группировки запросов от пользователя, чтобы сформировать одну сессию просмотра. Состояние сессии (*session state*) позволяет связывать данные с сессией. Оно идеально подходит для нашего класса Cart. Мы хотим, чтобы у каждого пользователя была своя корзина, и чтобы она сохранялась в промежутках времени между запросами. Данные, которые связываются с сессией, удаляются, когда сессия истекает (обычно потому, что пользователь не отправлял запросы некоторое время). Это означает, что мы не должны управлять хранением или жизненным циклом объектов Cart. Чтобы добавить объект в состояние сессии, мы устанавливаем значение для ключа в объекте Session, например:

```
Session["Cart"] = cart;
```

Чтобы извлечь объект снова, мы просто считываем тот же ключ, например:

```
Cart cart = (Cart)Session["Cart"];
```

#### *Совет*

*Объекты состояния сеанса хранятся в памяти сервера ASP.NET по умолчанию, но вы можете настроить различные другие подходы к хранению, в том числе с использованием базы данных SQL.*

---

Для методов AddToCart и RemoveFromCart мы использовали имена параметров, которые соответствуют элементам input в формах HTML, которые мы создали в представлении

`ProductSummary.cshtml`. Это позволяет MVC Framework связывать входящие переменные формы POST с этими параметрами, что избавляет нас от необходимости обрабатывать форму самим.

## Отображаем содержимое корзины

Последнее замечание по поводу контроллера `Cart` состоит в том, что и метод `AddToCart`, и `RemoveFromCart` вызывают метод `RedirectToAction`. В результате этого браузеру клиента отправляется HTTP-инструкция перенаправления, которая сообщает браузеру запросить новый URL. В этом случае мы сообщаем браузеру запросить URL, который будет вызывать метод действия `Index` контроллера `Cart`.

Мы реализуем метод `Index` и будем использовать его для отображения содержимого корзины. Если вы вернетесь к рисунку 8-8, то увидите, это наш рабочий поток после того, как пользователь нажимает кнопку `Add to cart`.

Нам нужно передать две порции информации в представление, которое будет отображать содержимое корзины: объект `Cart` и URL, который будет отображен, если пользователь нажмет кнопку `Continue shopping`. Для этого мы создадим простой класс модели представления. Создайте новый класс под названием `CartIndexViewModel` в папке `Models` проекта `SportsStore.WebUI`. Содержание этого класса показано в листинге 8-17.

### Листинг 8-17: Класс `CartIndexViewModel`

```
using SportsStore.Domain.Entities;

namespace SportsStore.WebUI.Models
{
    public class CartIndexViewModel
    {
        public Cart Cart { get; set; }
        public string ReturnUrl { get; set; }
    }
}
```

Когда у нас готова модель представления, мы можем реализовать метод действия `Index` в классе контроллера `Cart`, как показано в листинге 8-18.

### Листинг 8-18: Метод действия `Index`

```
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using SportsStore.WebUI.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace SportsStore.WebUI.Controllers
{
    public class CartController : Controller
    {
        private IProductRepository repository;

        public CartController(IProductRepository repo)
        {
            repository = repo;
        }

        public ViewResult Index(string returnUrl)
```

```

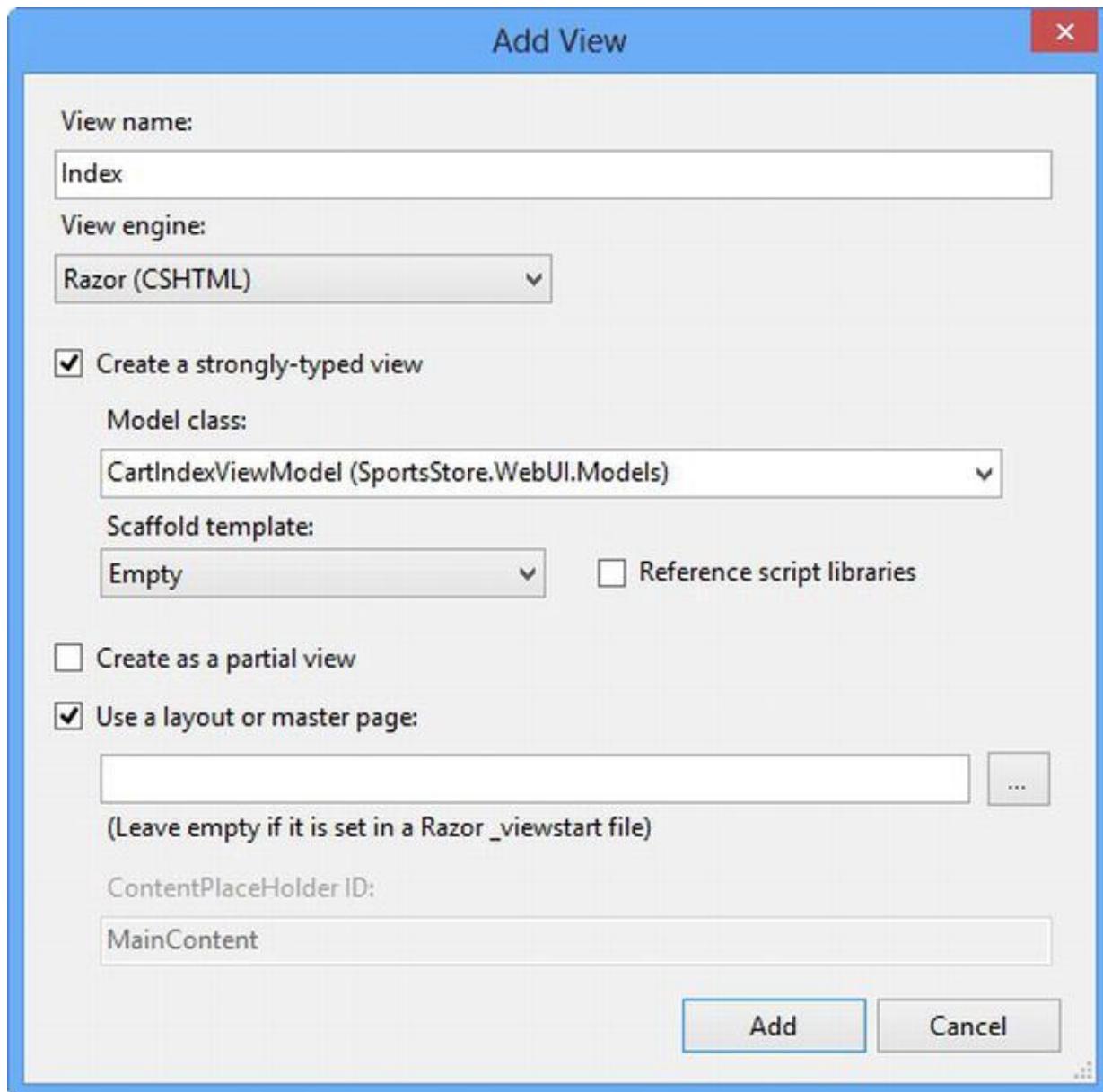
    {
        return View(new CartIndexViewModel
        {
            Cart = GetCart(),
            ReturnUrl = returnUrl
        });
    }

    // ...other action methods omitted for brevity...
}
}

```

Последнее, что нужно сделать, чтобы отобразить содержимое корзины, - это создать новое представление. Щелкните правой кнопкой мыши метод Index и выберите Add View из контекстного меню. Назовите представление Index, отметьте флажком опцию Create a strongly typed view и выберите CartIndexViewModel как класс модели, как показано на рисунке 8-9.

**Рисунок 8-9:** Добавляем представление Index



Мы хотим, чтобы содержимое корзины выглядело так же, как и остальные страницы приложения, так что убедитесь, что вы выбрали опцию Use a layout и оставили текстовое поле пустым, чтобы по умолчанию использовался файл \_Layout.cshtml. Нажмите кнопку Add, чтобы создать представление, и отредактируйте содержимое так, чтобы оно соответствовало листингу 8-19.

#### Листинг 8-19: Представление Index

```
@model SportsStore.WebUI.Models.CartIndexViewModel

 @{
    ViewBag.Title = "Sports Store: Your Cart";
}

<h2>Your cart</h2>
<table width="90%" align="center">
    <thead>
        <tr>
            <th align="center">Quantity</th>
            <th align="left">Item</th>
            <th align="right">Price</th>
            <th align="right">Subtotal</th>
        </tr>
    </thead>
    <tbody>
        @foreach (var line in Model.Cart.Lines) {
            <tr>
                <td align="center">@line.Quantity</td>
                <td align="left">@line.Product.Name</td>
                <td align="right">@line.Product.Price.ToString("c")</td>
                <td align="right">@((line.Quantity * line.Product.Price).ToString("c"))</td>
            </tr>
        }
    </tbody>
    <tfoot>
        <tr>
            <td colspan="3" align="right">Total:</td>
            <td align="right">
                @Model.Cart.ComputeTotalValue().ToString("c")
            </td>
        </tr>
    </tfoot>
</table>
<p align="center" class="actionButtons">
    <a href="@Model.ReturnUrl">Continue shopping</a>
</p>
```

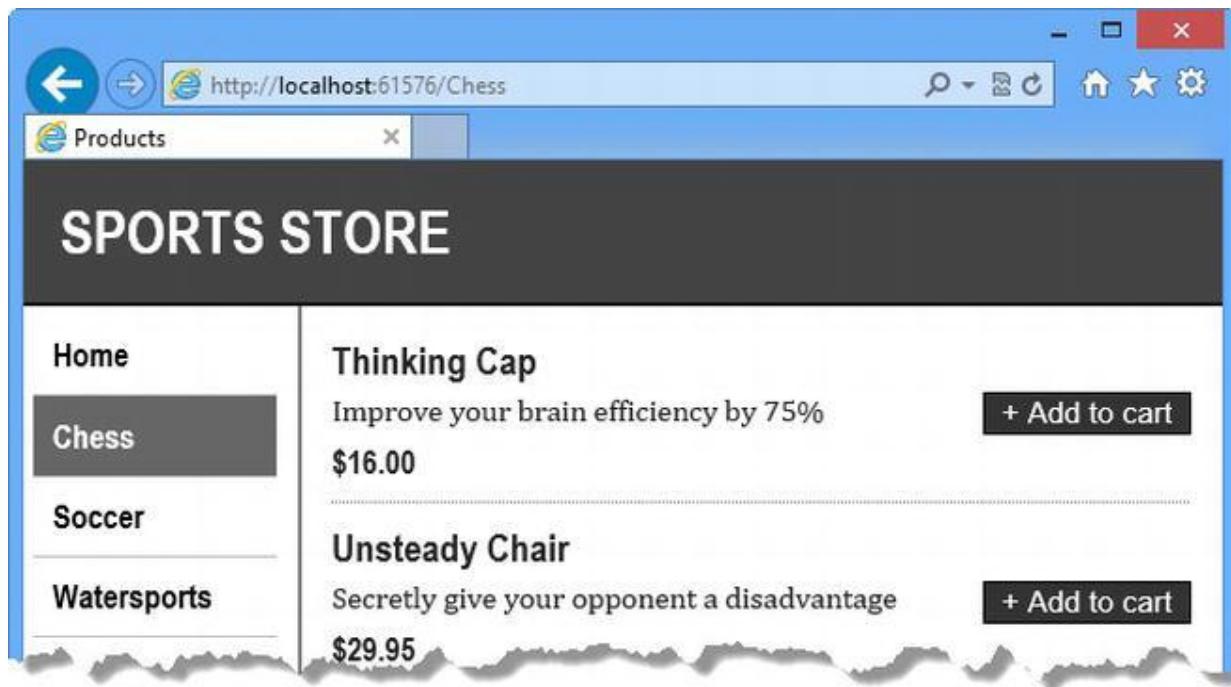
Представление выглядит сложнее, чем оно есть на самом деле. Оно просто перечисляет строки в корзине и добавляет ряды для каждой из них HTML-таблицу, а также общую стоимость в каждом ряду и общую стоимость всей корзины. Нам осталось добавить еще немного CSS. Добавьте стили, показанные в листинге 8-20, к файлу Site.css.

#### Листинг 8-20: CSS для отображения содержимого корзины

```
H2 { margin-top: 0.3em }
TFOOT TD { border-top: 1px dotted gray; font-weight: bold; }
.actionButtons A, INPUT.actionButtons {
    font: .8em Arial; color: White; margin: .5em;
    text-decoration: none; padding: .15em 1.5em .2em 1.5em;
    background-color: #353535; border: 1px solid black;
}
```

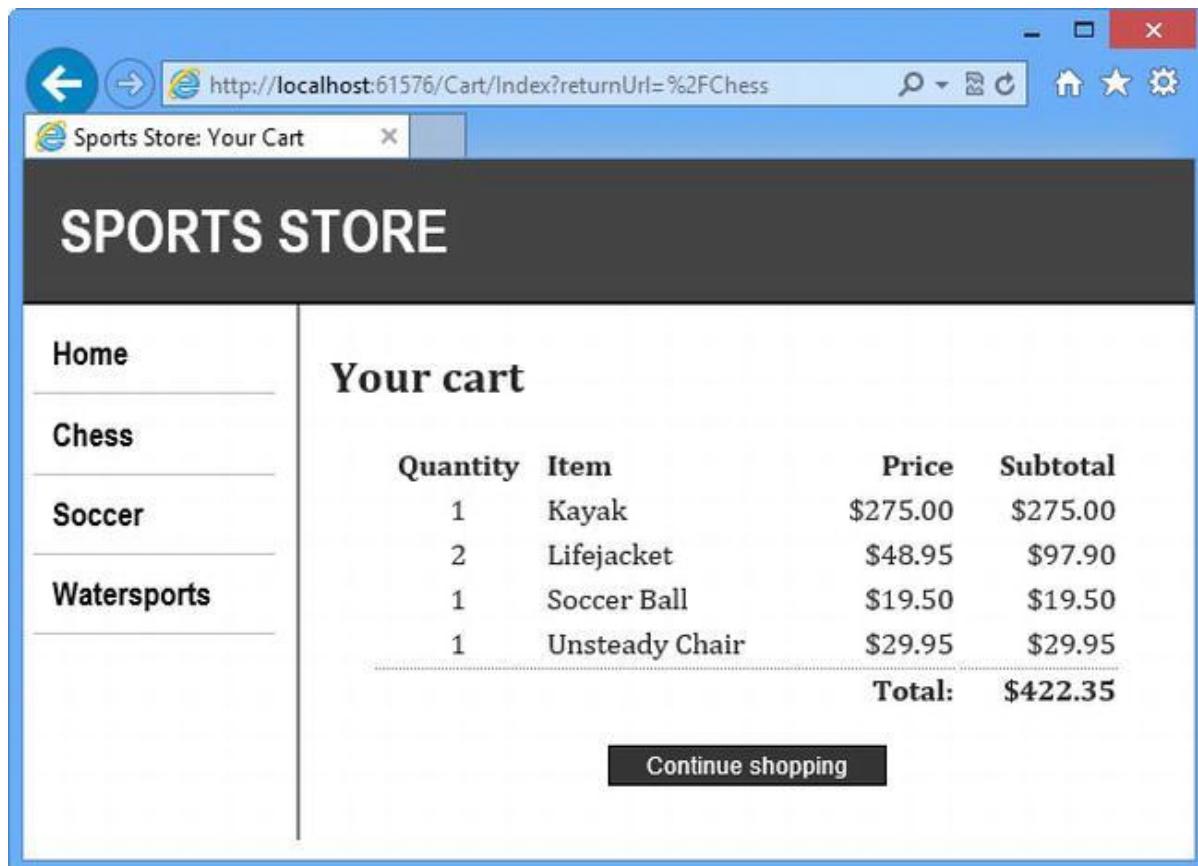
Теперь у нас готовы базовые функции корзины. Во-первых, товары выводятся с кнопкой для добавления в корзину, как показано на рисунке 8-10.

Рисунок 8-10: Кнопка Add to cart



Во-вторых, когда мы нажимаем кнопку Add to cart, соответствующий товар добавляется в корзину и отображаются общие сведения о корзине, как показано на рисунке 8-11. Мы можем нажать кнопку Continue shopping и вернуться на страницу товара, с которой пришли - все выглядит очень красиво и работает гладко.

Рисунок 8-11: Отображение содержимого корзины



## Резюме

В этой главе мы начали реализовывать части приложения SportsStore для работы пользователей. Мы предоставили средства, с помощью которых пользователь может перемещаться по категориям, и создали основные блоки для добавления товаров в корзину. Однако осталось еще много работы, которую мы продолжим в следующей главе.

# SportsStore: завершение функционала для корзины покупателя

Мы продолжим строить наше приложение SportsStore. В предыдущей главе мы добавили базовую поддержку корзины, а теперь собираемся улучшить и дополнить ее функциональность.

## Использование модели связывания данных

MVC Framework использует систему под названием *модель связывания данных* для создания объектов C# из HTTP-запросов и передачи их в качестве значений параметров в методы действий. Таким образом, например, MVC обрабатывает формы. Платформа смотрит на параметры целевого метода действия и использует *механизм связывания данных модели*, чтобы получить значения входных элементов формы и преобразовать их в одноименный тип параметра.

Механизмы связывания могут создавать типы C# из любых данных, доступных в запросе. Это является одной из центральных возможностей MVC Framework. Мы создадим пользовательский механизм связывания, чтобы улучшить наш класс `CartController`.

Нам нравится использовать состояние сеанса в контроллере `Cart` для хранения и управления объектами `Cart`, но нам не нравится то, как мы должны это делать. Это не соответствует всей остальной модели приложения, которая основана на параметрах методов действий. Мы не сможем должным образом протестировать класс `CartController`, если только не создадим имитацию параметра `Session` базового класса, а это означает, что нужно будет создавать имитацию класса `Controller` и много других вещей, с которыми мы бы предпочли не иметь дела.

Чтобы решить эту проблему, мы создадим пользовательский механизм связывания, который будет получать объект `Cart`, содержащийся в данных сессии. Тогда MVC Framework сможет создавать объекты `Cart` и передавать их в качестве параметров методов действий в наш класс `CartController`. Связывание данных – очень мощная и гибкая возможность. Мы рассмотрим ее подробнее в главе 22, но этот пример отлично подходит, чтобы с ней познакомиться.

### Создаем пользовательский механизм связывания данных

Мы создаем пользовательский механизм связывания путем реализации интерфейса `IModelBinder`. Создайте новую папку под названием `Binders` в проекте `SportsStore.WebUI`, а в ней - класс `CartModelBinder`. Определение класса `CartModelBinder` показано в листинге 9-1.

#### Листинг 9-1: Класс `CartModelBinder`

```
using System;
using System.Web.Mvc;
using SportsStore.Domain.Entities;
namespace SportsStore.WebUI.Binders
{
    public class CartModelBinder : IModelBinder
    {
        private const string sessionKey = "Cart";

        public object BindModel(ControllerContext controllerContext, ModelBindingContext bindingContext)
        {
            // get the Cart from the session
            Cart cart = (Cart)controllerContext.HttpContext.Session[sessionKey];
```

```

    // create the Cart if there wasn't one in the session data
    if (cart == null)
    {
        cart = new Cart();
        controllerContext.HttpContext.Session[sessionKey] = cart;
    }

    // return the cart
    return cart;
}
}
}

```

Интерфейс `IModelBinder` определяет один метод: `BindModel`. Мы передаем в него два параметра для того, чтобы сделать возможным создание объекта доменной модели. `ControllerContext` обеспечивает доступ ко всей информации, которой располагает класс контроллера и которая включает в себя детали запроса клиента. `ModelBindingContext` предоставляет информацию об объекте создаваемой модели, и некоторые инструменты, которые облегчат процесс связывания. Мы вернемся к этому классу в главе 22.

Класс `ControllerContext` интересует нас больше всего. У него есть свойство `HttpContext`, у которого, в свою очередь, есть свойство `Session`, которое позволяет получать и устанавливать данные сессии. Прочитав значение ключа из данных сессии, мы получаем объект `Cart` или, если его еще не существует, создаем новый.

Мы должны сообщить MVC Framework, что она может использовать класс `CartModelBinder` для создания экземпляров объекта `Cart`. Мы делаем это в методе `Application_Start` файла `Global.asax`, как показано в листинге 9-2.

#### Листинг 9-2: Регистрируем класс `CartModelBinder`

```

using SportsStore.Domain.Entities;
using SportsStore.WebUI.Binders;
using SportsStore.WebUI.Infrastructure;
using System.Web.Http;
using System.Web.Mvc;
using System.Web.Optimization;
using System.Web.Routing;

namespace SportsStore.WebUI
{
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();
            WebApiConfig.Register(GlobalConfiguration.Configuration);
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);

            ControllerBuilder.Current.SetControllerFactory(new NinjectControllerFactory());
            ModelBinders.Binders.Add(typeof(Cart), new CartModelBinder());
        }
    }
}

```

Теперь мы можем обновить класс `CartController`, чтобы удалить метод `GetCart` и задействовать на наш механизм связывания, который MVC Framework будет применять автоматически. Изменения показаны в листинге 9-3.

### Листинг 9-3: Используем механизм связывания в `CartController`

```
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using SportsStore.WebUI.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace SportsStore.WebUI.Controllers
{
    public class CartController : Controller
    {
        private IProductRepository repository;

        public CartController(IProductRepository repo)
        {
            repository = repo;
        }

        public ViewResult Index(Cart cart, string returnUrl)
        {
            return View(new CartIndexViewModel
            {
                Cart = cart,
                ReturnUrl = returnUrl
            });
        }

        public RedirectToRouteResult AddToCart(Cart cart, int productId, string returnUrl)
        {
            Product product = repository.Products.FirstOrDefault(p => p.ProductID == productId);
            if (product != null)
            {
                cart.AddItem(product, 1);
            }
            return RedirectToAction("Index", new { returnUrl });
        }

        public RedirectToRouteResult RemoveFromCart(Cart cart, int productId, string returnUrl)
        {
            Product product = repository.Products.FirstOrDefault(p => p.ProductID == productId);
            if (product != null)
            {
                cart.RemoveLine(product);
            }
            return RedirectToAction("Index", new { returnUrl });
        }
    }
}
```

Мы удалили метод `GetCart` и добавили параметр `Cart` в каждый метод действия. Когда MVC Framework получает запрос, который требует, скажем, вызвать метод `AddToCart`, она будет сначала смотреть на параметры для метода действия. Она рассмотрит список доступных механизмов

связывания и попытается найти тот, который сможет создать экземпляры каждого типа параметра. Наш пользовательский механизм связывания должен будет создать объект `Cart`, что он и сделает, используя состояние сеанса. Между обращениями к нашему механизму связывания и механизму по умолчанию MVC Framework может создать набор параметров, которые необходимы для вызова метода действия, позволяя нам реорганизовать контроллер так, чтобы в нем не осталось сведений о том, как создаются объекты `Cart` при получении запросов.

Использование подобного механизма связывания дает нам несколько преимуществ. Первое заключается в том, что мы отделили логику для создания объектов `Cart` от контроллера, что позволит нам изменять способ сохранения этих объектов без необходимости изменять контроллер. Вторым преимуществом является то, что любой класс контроллера, который работает с объектами `Cart`, может просто объявить их как параметры метода действия и воспользоваться пользовательским механизмом связывания. Третье и, на наш взгляд, самое главное преимущество состоит в том, что теперь мы сможем тестировать контроллер `Cart`, не создавая имитаций встроенных возможностей ASP.NET.

### Модульный тест: контроллер `Cart`

Мы можем протестировать класс `CartController`, создавая объекты `Cart` и передавая их в методы действия. Мы хотим проверить три аспекта этого контроллера:

- Метод `AddToCart` должен добавить выбранный товар в корзину покупателя.
- После добавления товара в корзину он должен перенаправить нас в представление `Index`.
- URL, по которому пользователь сможет вернуться в каталог, должен быть корректно передан в метод действия `Index`.

Вот модульные тесты, которые мы добавили в файл `CartTests.cs` проекта `SportsStore.UnitTests`:

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using SportsStore.Domain.Entities;
using System.Linq;
using Moq;
using SportsStore.Domain.Abstract;
using SportsStore.WebUI.Controllers;
using System.Web.Mvc;
using SportsStore.WebUI.Models;
namespace SportsStore.UnitTests
{
    [TestClass]
    public class CartTests
    {
        //...existing test methods omitted for brevity...
        [TestMethod]
        public void Can_Add_To_Cart()
        {
            // Arrange - create the mock repository
            Mock<IProductRepository> mock = new Mock<IProductRepository>();
            mock.Setup(m => m.Products).Returns(new Product[] {
                new Product {ProductID = 1, Name = "P1", Category = "Apples"},
            }.AsQueryable());

            // Arrange - create a Cart
            Cart cart = new Cart();

            // Arrange - create the controller
            CartController target = new CartController(mock.Object);
```

```

// Act - add a product to the cart
target.AddToCart(cart, 1, null);

// Assert
Assert.AreEqual(cart.Lines.Count(), 1);
Assert.AreEqual(cart.Lines.ToArray()[0].Product.ProductID, 1);
}

[TestMethod]
public void Adding_Product_To_Cart_Goes_To_Cart_Screen()
{
    // Arrange - create the mock repository
    Mock<IProductRepository> mock = new Mock<IProductRepository>();
    mock.Setup(m => m.Products).Returns(new Product[] {
        new Product { ProductID = 1, Name = "P1", Category = "Apples" },
    }.AsQueryable());

    // Arrange - create a Cart
    Cart cart = new Cart();

    // Arrange - create the controller
    CartController target = new CartController(mock.Object);

    // Act - add a product to the cart
    RedirectToRouteResult result = target.AddToCart(cart, 2, "myUrl");

    // Assert
    Assert.AreEqual(result.RouteValues["action"], "Index");
    Assert.AreEqual(result.RouteValues["returnUrl"], "myUrl");
}

[TestMethod]
public void Can_View_Cart_Contents()
{
    // Arrange - create a Cart
    Cart cart = new Cart();

    // Arrange - create the controller
    CartController target = new CartController(null);

    // Act - call the Index action method
    CartIndexViewModel result = (CartIndexViewModel)target.Index(cart,
    "myUrl").ViewData.Model;

    // Assert
    Assert.AreSame(result.Cart, cart);
    Assert.AreEqual(result.ReturnUrl, "myUrl");
}
}
}

```

# Завершаем функционал для корзины

Теперь, когда пользовательский механизм связывания данных создан, мы можем завершить функциональность корзины, добавив две новые функции. Первая позволит пользователю удалять товар из корзины. Вторая будет отображать виджет корзины в верхней части страницы.

## Удаление товаров из корзины

Мы уже определили и проверили метод действия `RemoveFromCart` в контроллере. Чтобы позволить пользователю удалять товары, нужно только сделать этот метод доступным в представлении, для чего мы добавим кнопку `Remove` в каждую строку представления для корзины. Изменения в `Views/Cart/Index.cshtml` показаны в листинге 9-4.

**Листинг 9-4:** Создаем кнопку Remove

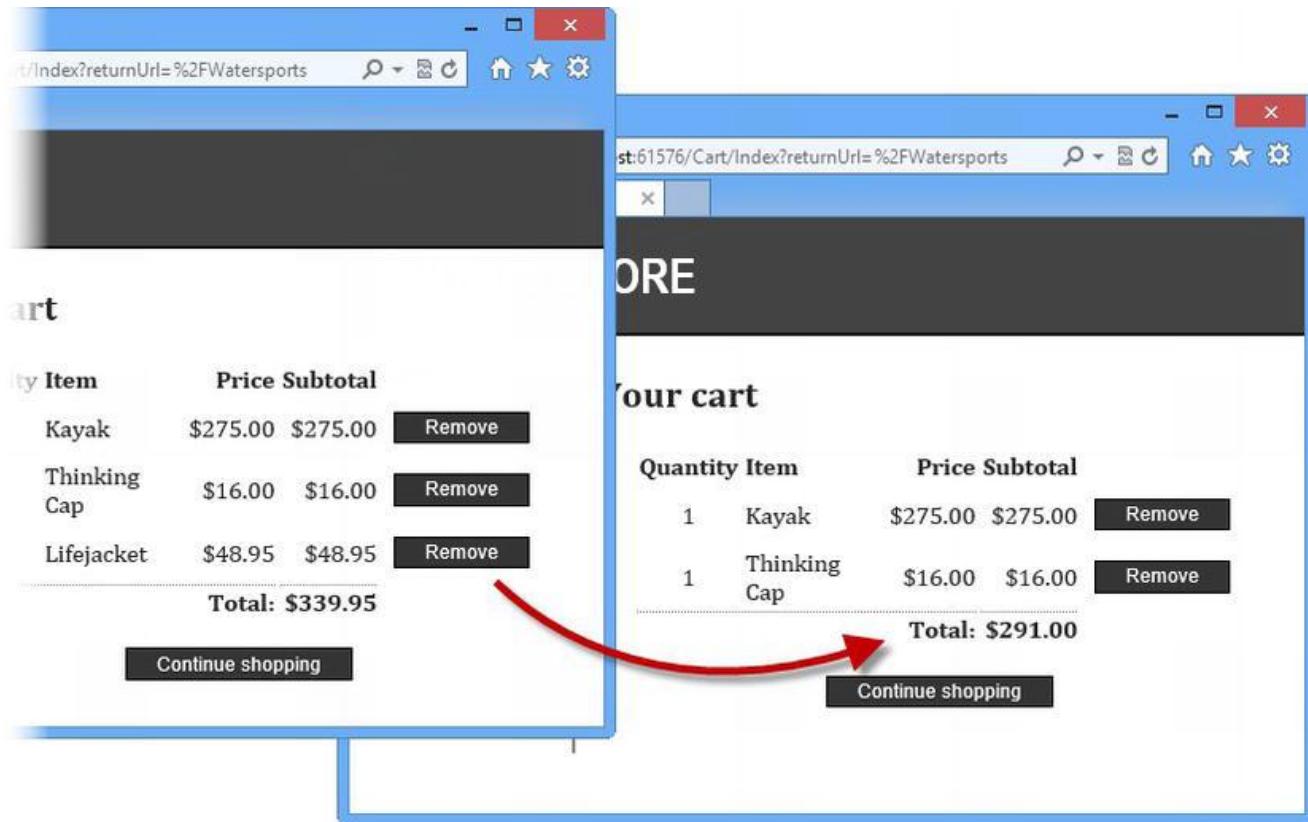
```
<tbody>
@foreach (var line in Model.Cart.Lines) {
    <tr>
        <td align="center">@line.Quantity</td>
        <td align="left">@line.Product.Name</td>
        <td align="right">@line.Product.Price.ToString("c")</td>
        <td align="right">@(line.Quantity * line.Product.Price).ToString("c")</td>
        <td>
            @using (Html.BeginForm("RemoveFromCart", "Cart")) {
                @Html.Hidden("ProductId", line.Product.ProductID)
                @Html.HiddenFor(x => x.ReturnUrl)
                <input class="actionButtons" type="submit" value="Remove" />
            }
        </td>
    </tr>
}
</tbody>
```

### Примечание

*Мы можем использовать строго типизированный вспомогательный метод `Html.HiddenFor`, чтобы создать скрытое поле для свойства модели `ReturnUrl`, но мы должны использовать строко-ориентированный вспомогательный метод `Html.Hidden`, чтобы сделать то же самое для поля `Product ID`. Если мы запишем `Html.HiddenFor(x => line.Product.ProductID)`, вспомогательный метод визуализирует скрытое поле под названием `line.Product.ProductID`. Имя поля не будет совпадать с именами параметров метода действия `CartController.RemoveFromCart`, из-за чего механизмы связывания по умолчанию не будут работать и MVC Framework не сможет вызвать метод.*

Вы можете проверить работу кнопок `Remove`, если запустите приложение, добавите какие-нибудь товары в корзину и кликните по одной из них. Результат показан на рисунке 9-1.

**Рисунок 9-1:** Удаление товара из корзины



## Добавляем виджет корзины

У нас уже готова функционирующая корзина, но еще есть проблема с тем, как она интегрирована в интерфейс. Пользователи могут просмотреть ее содержимое, только открыв страницу корзины. А открыть страницу корзины они смогут, только добавив в нее новый товар.

Чтобы решить эту проблему, мы добавим виджет, который выводит краткую информацию о корзине и при нажатии отобразит ее содержимое. Мы сделаем это почти также, как добавляли виджет навигации - используя действие, вывод которого будет внедрен в макет Razor.

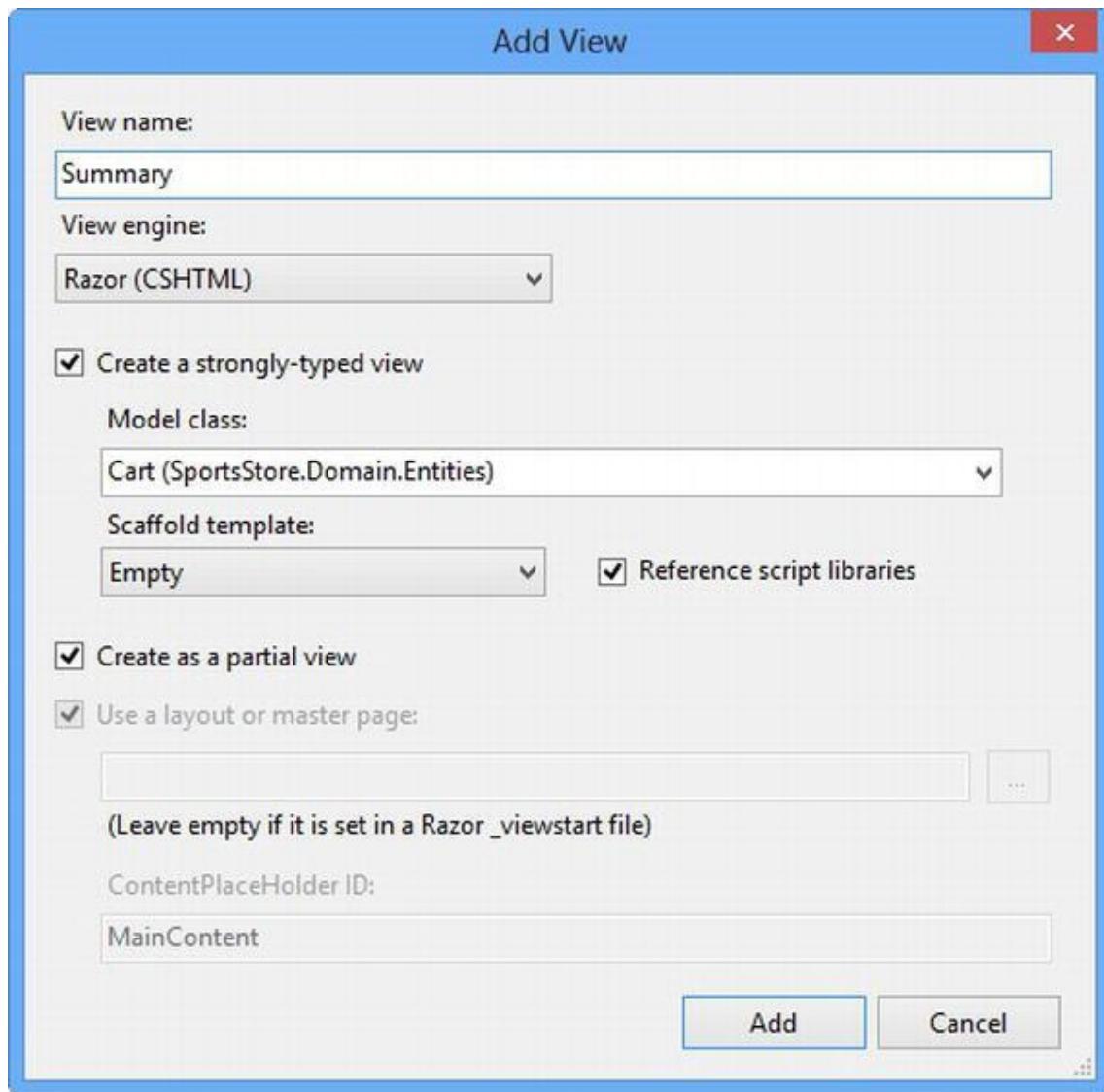
Для начала нам нужно добавить в класс `CartController` простой метод, показанный в листинге 9-5.

### Листинг 9-5: Добавляем метод `Summary` в контроллер `Cart`

```
public PartialViewResult Summary(Cart cart) {
    return PartialView(cart);
}
```

Как видите, это очень простой метод. Он просто визуализирует представление, предоставляя текущий объект `Cart` (который будет получен с помощью нашего пользовательского механизма связывания), в качестве данных представления. Нам нужно создать частичное представление, которое будет отображаться в ответ на вызов метода `Summary`. Кликните правой кнопкой мыши метод `Summary` и выберите `Add View` из контекстного меню. Назовите представление `Summary`, отметьте опцию `Create as strongly typed view` и установите класс модели `Cart`, как показано на рисунке 9-2. Мы хотим внедрить частичное представление в страницу, так что отметьте опцию `Create as a partial view`.

**Рисунок 9-2:** Добавляем представление Summary



Измените новое частичное представление так, чтобы оно соответствовало листингу 9-6.

**Листинг 9-6:** Частичное представление Summary

```
@model SportsStore.Domain.Entities.Cart

<div id="cart">
    <span class="caption">
        <b>Your cart:</b>
        @Model.Lines.Sum(x => x.Quantity) item(s),
        @Model.ComputeTotalValue().ToString("c")
    </span>

    @Html.ActionLink("Checkout", "Index", "Cart", new { returnUrl =
Request.Url.PathAndQuery }, null)
</div>
```

Это простое представление, которое отображает количество товаров в корзине, их общую стоимость и ссылку, по которой можно просмотреть содержимое корзины. Теперь, когда мы определили представление, которое возвращает метод действия `Summary`, мы можем включить результат рендеринга в файл `_Layout.cshtml`, как показано в листинге 9-7.

### Листинг 9-7: Добавляем частичное представление Summary в макет

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    <link href="~/Content/Site.css" type="text/css" rel="stylesheet" />
</head>
<body>
    <div id="header">
        @Html.RenderAction("Summary", "Cart");
        <div class="title">SPORTS STORE</div>
    </div>
    <div id="categories">
        @Html.RenderAction("Menu", "Nav");
    </div>
    <div id="content">
        @RenderBody()
    </div>
</body>
</html>
```

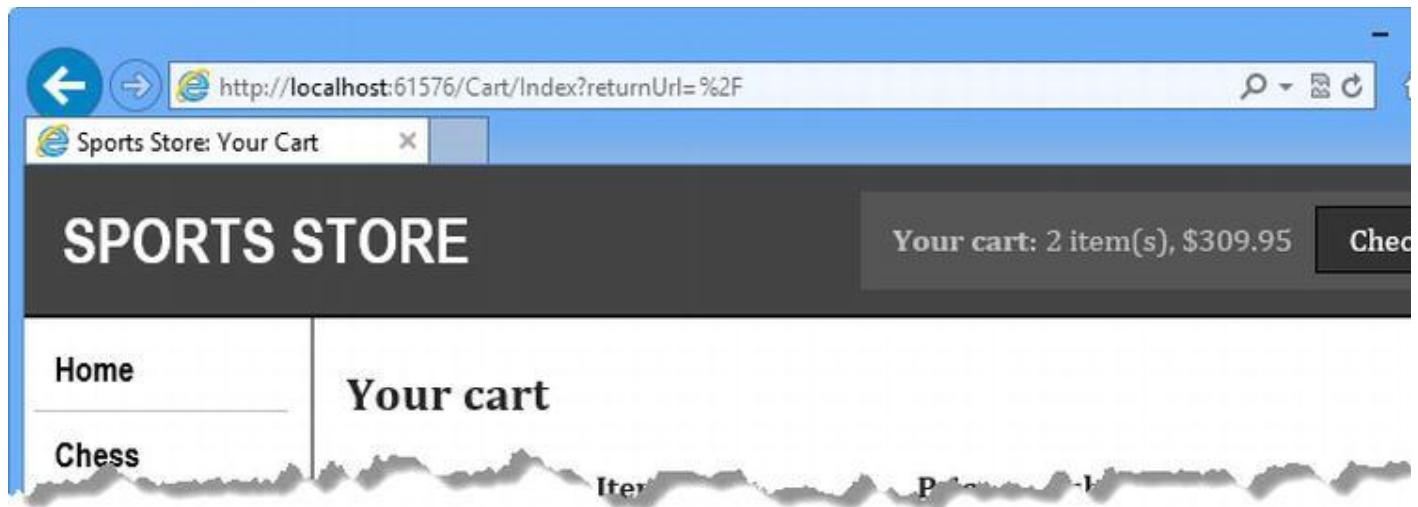
Напоследок мы добавим некоторые дополнительные правила CSS для форматирования элементов в частичное представление. Добавьте стили из листинга 9-8 в файл Site.css проекта SportsStore.WebUI.

### Листинг 9-8: Добавляем стили в Site.css

```
DIV#cart { float:right; margin: .8em; color: Silver;
    background-color: #555; padding: .5em .5em .5em 1em; }
DIV#cart A { text-decoration: none; padding: .4em 1em .4em 1em; line-height:2.1em;
    margin-left: .5em; background-color: #333; color:White; border: 1px solid black; }
```

Запустив приложение, вы можете увидеть виджет корзины. Количество элементов и их общая стоимость увеличивается, когда вы добавляете товар в корзину, как показано на рисунке 9-3.

Рисунок 9-3: Виджет корзины



Благодаря этому дополнению пользователи будут знать, что у них в корзине, и использовать очевидный способ для перехода к оплате. Вы еще раз можете убедиться, как легко использовать RenderAction для внедрения вывода метода действия в страницу. Эта техника также позволяет разделить функциональность приложения на отдельные многократно используемые блоки.

# Отправка заказов

Мы дошли до заключительной функции для работы пользователей в SportsStore: возможности подтвердить заказ и завершить покупку. В следующих разделах мы расширим нашу доменную модель так, чтобы обеспечить поддержку ввода реквизитов доставки, и добавим функцию для обработки этой информации.

## Расширяем доменную модель

Добавьте класс под названием `ShippingDetails` в папку `Entities` проекта `SportsStore.Domain`. Это класс, который мы будем использовать для представления клиенту полей для ввода информации о доставке. Его содержимое показано в листинге 9-9.

### Листинг 9-9: Класс `ShippingDetails`

```
using System.ComponentModel.DataAnnotations;

namespace SportsStore.Domain.Entities
{
    public class ShippingDetails
    {
        [Required(ErrorMessage = "Please enter a name")]
        public string Name { get; set; }

        [Required(ErrorMessage = "Please enter the first address line")]
        public string Line1 { get; set; }
        public string Line2 { get; set; }
        public string Line3 { get; set; }

        [Required(ErrorMessage = "Please enter a city name")]
        public string City { get; set; }

        [Required(ErrorMessage = "Please enter a state name")]
        public string State { get; set; }
        public string Zip { get; set; }

        [Required(ErrorMessage = "Please enter a country name")]
        public string Country { get; set; }
        public bool GiftWrap { get; set; }
    }
}
```

Как вы видите из листинга 9-9, мы используем атрибуты валидации из пространства имен `System.ComponentModel.DataAnnotations`, как мы это уже делали в главе 2. Валидация будет рассмотрена подробно в главе 23.

### Примечание

*В классе `ShippingDetails` нет никакой функциональности, так что он не нуждается в тестировании.*

## Добавляем процесс подтверждения заказа

Мы хотим реализовать функционал до того пункта, где пользователи смогут ввести информацию о доставке и отправить заказ. Чтобы это выполнить, для начала нам нужно добавить кнопку `Checkout`

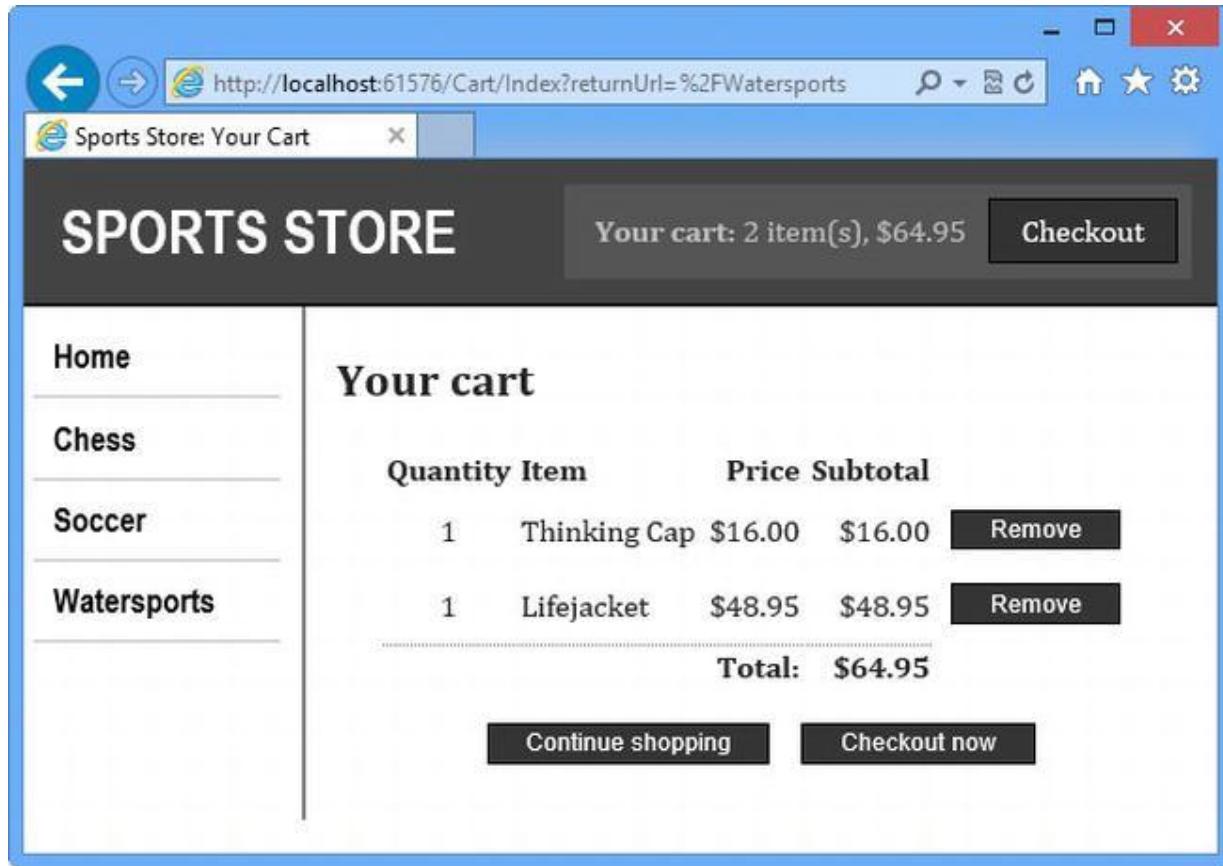
now в представление для корзины. В листинге 9-10 показано изменение, которое мы должны внести в файл Views/Cart/Index.cshtml.

#### Листинг 9-10: Добавляем кнопку Checkout now

```
</table>
<p align="center" class="actionButtons">
    <a href="@Model.ReturnUrl">Continue shopping</a>
    @Html.ActionLink("Checkout now", "Checkout")
</p>
```

Это единственное изменение генерирует ссылку, нажатие на которую вызывает метод действия Checkout контроллера Cart. Вы можете увидеть эту кнопку на рисунке 9-4.

Рисунок 9-4: Кнопка Checkout Now



Как и следовало ожидать, теперь нам нужно определить метод Checkout в классе CartController, как показано в листинге 9-11.

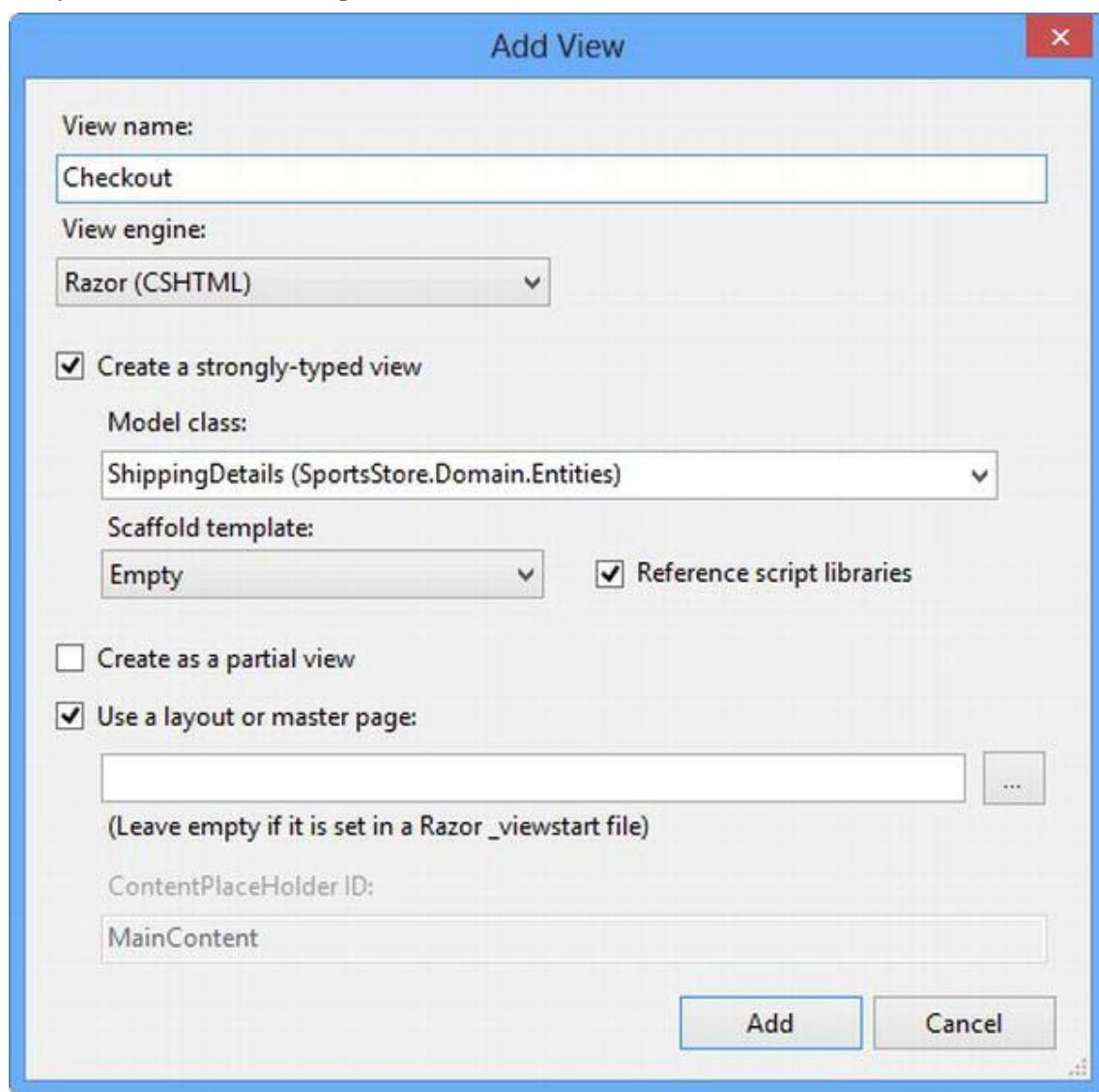
#### Листинг 9-11: Метод действия Checkout

```
public ViewResult Checkout() {
    return View(new ShippingDetails());
}
```

Метод Checkout возвращает представление по умолчанию и передает в него новый объект ShippingDetails как модель представления. Чтобы создать соответствующее представление, кликните правой кнопкой мыши метод Checkout, выберите Add View и заполните диалоговое окно, как показано на рисунке 9-5. Мы собираемся использовать доменный класс ShippingDetails как

основу для строго типизированного представления. Отметьте флажком опцию Use a layout, потому что мы визуализируем целую страницу и хотим, чтобы она выглядела также, как и все приложение.

**Рисунок 9-5:** Добавляем представление Checkout



Приведите содержимое представления в соответствие с разметкой, показанной в листинге 9-12.

**Листинг 9-12:** Представление Checkout.cshtml

```
@model SportsStore.Domain.Entities.ShippingDetails

@{
    ViewBag.Title = "SportStore: Checkout";
}

<h2>Check out now</h2>
Please enter your details, and we'll ship your goods right away!
@using (Html.BeginForm())
{
    <h3>Ship to</h3>
    <div>Name: @Html.EditorFor(x => x.Name)</div>
    <h3>Address</h3>
    <div>Line 1: @Html.EditorFor(x => x.Line1)</div>
    <div>Line 2: @Html.EditorFor(x => x.Line2)</div>
```

```

<div>Line 3: @Html.EditorFor(x => x.Line3)</div>
<div>City: @Html.EditorFor(x => x.City)</div>
<div>State: @Html.EditorFor(x => x.State)</div>
<div>Zip: @Html.EditorFor(x => x.Zip)</div>
<div>Country: @Html.EditorFor(x => x.Country)</div>
<h3>Options</h3>
<label>
    @Html.EditorFor(x => x.GiftWrap)
    Gift wrap these items
</label>

<p align="center">
    <input class="actionButtons" type="submit" value="Complete order" />
</p>
}

```

Вы увидите, как визуализируется это представление, запустив приложение, добавив товар в корзину и нажав кнопку `Checkout now`. На рисунке 9-6 показано, что представление отображается в виде формы для ввода реквизитов доставки.

**Рисунок 9-6:** Форма для реквизитов доставки

The screenshot shows a web browser window for the "SportStore" website. The URL in the address bar is `http://localhost:61576/Cart/Checkout`. The page title is "SportStore: Checkout". The main content area is titled "SPORTS STORE" and displays a message: "Your cart: 2 item(s), \$323.95" and a "Checkout" button. On the left, there's a sidebar with navigation links: "Home", "Chess", "Soccer", and "Watersports". The main form area starts with "Check out now" and a note: "Please enter your details, and we'll ship your goods right away!". It has a "Ship to" section with fields for Name (input), Line 1 (input), Line 2 (input), Line 3 (input), City (input), State (input), Zip (input), and Country (input). Below that is an "Address" section with the same set of input fields. At the bottom of the form is an "Options" section containing a checkbox labeled "Gift wrap these items" and a "Complete order" button.

Мы визуализировали элементы `input` для каждого из полей формы с помощью вспомогательного метода `Html.EditorFor`. Это пример шаблонного вспомогательного метода. Мы позволяем MVC Framework решать, какой элемент `input` требуется для свойства модели представления, а не указываем его явно (например, с помощью `Html.TextBoxFor`).

Мы подробно опишем шаблонные вспомогательные методы в главе 20, но вы уже можете видеть из рисунка, MVC достаточно умная платформа, чтобы визуализировать чекбокс для свойств `bool` (например, для опции `Gift wrap`) и текстовые поля для строковых свойств.

#### *Совет*

*Мы могли бы пойти еще дальше и заменить большую часть разметки представления на вызов вспомогательного метода `Html.EditorForModel`, который генерирует метки и поля ввода для всех свойств в классе модели представления `ShippingDetails`. Тем не менее, мы хотели разделить элементы так, чтобы поля для имени, адреса и дополнительных опций находились в разных частях формы, поэтому мы просто обращаемся к каждому свойству напрямую.*

## Создаем `IOrderProcessor`

В приложении нам нужен компонент, который будет обрабатывать детали заказа. Следуя принципам модели MVC, мы собираемся определить для этой функциональности интерфейс, написать его реализацию, а затем ассоциировать их с помощью нашего DI-контейнера Ninject.

## Определяем интерфейс

Добавьте новый интерфейс под названием `IOrderProcessor` в папку `Abstract` проекта `SportsStore.Domain` и отредактируйте его содержимое так, чтобы оно соответствовало листингу 9-13.

### Листинг 9-13: Интерфейс `IOrderProcessor`

```
using SportsStore.Domain.Entities;

namespace SportsStore.Domain.Abstract
{
    public interface IOrderProcessor
    {
        void ProcessOrder(Cart cart, ShippingDetails shippingDetails);
    }
}
```

## Создаем реализацию интерфейса

Наша реализация `IOrderProcessor` будет обрабатывать заказы, отправляя их по электронной почте администратору сайта. Конечно, мы упрощаем процесс продажи. Большинство сайтов электронной коммерции не ограничиваются подтверждением заказов по e-mail, к тому же мы не обеспечили поддержку для обработки кредитных карт и других форм оплаты. Но мы не хотим отвлекаться от MVC, так что будем работать с электронной почтой.

Создайте новый класс под названием `EmailOrderProcessor` в папке `Concrete` проекта `SportsStore.Domain` и отредактируйте содержимое так, чтобы он соответствовал листингу 9-14. Для отправки e-mail этот класс использует встроенную поддержку SMTP, которая включена в библиотеку .NET Framework.

**Листинг 9-14:** Класс EmailOrderProcessor

```
using System.Net.Mail;
using System.Text;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using System.Net;

namespace SportsStore.Domain.Concrete
{
    public class EmailSettings
    {
        public string MailToAddress = "orders@example.com";
        public string MailFromAddress = "sportsstore@example.com";
        public bool UseSsl = true;
        public string Username = "MySmtpUsername";
        public string Password = "MySmtpPassword";
        public string ServerName = "smtp.example.com";
        public int ServerPort = 587;
        public bool WriteAsFile = false;
        public string FileLocation = @"c:\sports_store_emails";
    }

    public class EmailOrderProcessor : IOrderProcessor
    {
        private EmailSettings emailSettings;
        public EmailOrderProcessor(EmailSettings settings)
        {
            emailSettings = settings;
        }

        public void ProcessOrder(Cart cart, ShippingDetails shippingInfo)
        {
            using (var smtpClient = new SmtpClient())
            {
                smtpClient.EnableSsl = emailSettings.UseSsl;
                smtpClient.Host = emailSettings.ServerName;
                smtpClient.Port = emailSettings.ServerPort;
                smtpClient.UseDefaultCredentials = false;
                smtpClient.Credentials = new NetworkCredential(emailSettings.Username,
emailSettings.Password);
                if (emailSettings.WriteAsFile)
                {
                    smtpClient.DeliveryMethod = SmtpDeliveryMethod.SpecifiedPickupDirectory;
                    smtpClient.PickupDirectoryLocation = emailSettings.FileLocation;
                    smtpClient.EnableSsl = false;
                }
                StringBuilder body = new StringBuilder()
                    .AppendLine("A new order has been submitted")
                    .AppendLine("----")
                    .AppendLine("Items:");

                foreach (var line in cart.Lines)
                {
                    var subtotal = line.Product.Price * line.Quantity;
                    body.AppendFormat("{0} x {1} (subtotal: {2:c}", line.Quantity,
line.Product.Name, subtotal);
                }

                body.AppendFormat("Total order value: {0:c}", cart.ComputeTotalValue())
                    .AppendLine("----")
                    .AppendLine("Ship to:")
                    .AppendLine(shippingInfo.Name)
                    .AppendLine(shippingInfo.Line1)
                    .AppendLine(shippingInfo.Line2 ?? "");
            }
        }
    }
}
```

```
        .AppendLine(shippingInfo.Line3 ?? "")
        .AppendLine(shippingInfo.City)
        .AppendLine(shippingInfo.State ?? "")
        .AppendLine(shippingInfo.Country)
        .AppendLine(shippingInfo.Zip)
        .AppendLine(" --- ")
        .AppendFormat("Gift wrap: {0}", shippingInfo.GiftWrap ? "Yes" : "No");

    MailMessage mailMessage = new MailMessage(
        emailSettings.MailFromAddress, // From
        emailSettings.MailToAddress, // To
        "New order submitted!", // Subject
        body.ToString()); // Body

    if (emailSettings.WriteAsFile)
    {
        mailMessage.BodyEncoding = Encoding.ASCII;
    }
    smtpClient.Send(mailMessage);
}
}
```

Для простоты мы также определили в листинге 9-14 класс `EmailSettings`. Экземпляр этого класса требуется конструктором `EmailOrderProcessor` и содержит все настройки, которые необходимы для конфигурации классов .NET, работающих с электронной почтой.

Совет

*Не беспокойтесь, если у вас нет сервера SMTP. Если вы установите свойству `EmailSettings.WriteAsFile` значение `true`, e-mail сообщения будут записываться как файлы в директорию, указанную в свойстве `FileLocation`. Эта директория должна существовать и быть доступной для записи. Файлы будут записаны с расширением `.eml`, но их можно прочитать в любом текстовом редакторе.*

## **Регистрируем реализацию**

Теперь у нас есть реализация интерфейса `IOrderProcessor` и средства для ее настройки, мы также можем использовать `Ninject` для создания ее экземпляров. Отредактируйте класс `NinjectControllerFactory` проекта `SportsStore.WebUI` и внесите в метод `AddBindings` изменения, показанные в листинге 9-15.

**Листинг 9-15:** Добавление привязок Ninject для IOrderProcessor

```
using System;
using System.Web.Mvc;
using System.Web.Routing;
using Ninject;
using SportsStore.Domain.Entities;
using SportsStore.Domain.Abstract;
using System.Collections.Generic;
using System.Linq;
using Moq;
using SportsStore.Domain.Concrete;
using System.Configuration;

namespace SportsStore.WebUI.Infrastructure
{
```

```

public class NinjectControllerFactory : DefaultControllerFactory
{
    private IKernel ninjectKernel;

    public NinjectControllerFactory()
    {
        ninjectKernel = new StandardKernel();
        AddBindings();
    }

    protected override IController GetControllerInstance(RequestContext requestContext,
Type controllerType)
    {
        return controllerType == null
            ? null
            : (IController)ninjectKernel.Get(controllerType);
    }

    private void AddBindings()
    {
        ninjectKernel.Bind<IProductRepository>().To<EFProductRepository>();
        EmailSettings emailSettings = new EmailSettings
        {
            WriteAsFile = bool.Parse(ConfigurationManager
                .AppSettings["Email.WriteAsFile"] ?? "false")
        };
        ninjectKernel.Bind<IOrderProcessor>()
            .To<EmailOrderProcessor>()
            .WithConstructorArgument("settings", emailSettings);
    }
}
}

```

Мы создали объект `EmailSettings`, который будем использовать с методом `Ninject WithConstructorArgument`. Он будет внедряться в конструктор `EmailOrderProcessor`, когда создаются новые экземпляры для обслуживания запросов интерфейса `IOrderProcessor`. В листинге 9-15 мы установили значение только для одного из свойств `EmailSettings` - `WriteAsFile`. Мы читаем значение этого свойства с помощью свойства `ConfigurationManager.AppSettings`, которое дает нам доступ к настройкам приложения в файле `Web.config` (в корневой папке проекта), которые показаны в листинге 9-16.

#### Листинг 9-16: Настройки приложения в файле `Web.config`

```

<appSettings>
    <add key="webpages:Version" value="2.0.0.0" />
    <add key="webpages:Enabled" value="false" />
    <add key="PreserveLoginUrl" value="true" />
    <add key="ClientValidationEnabled" value="true" />
    <add key="UnobtrusiveJavaScriptEnabled" value="true" />
    <add key="Email.WriteAsFile" value="true"/>
</appSettings>

```

### Завершаем `CartController`

Для завершения класса `CartController` мы должны изменить конструктор так, что он запрашивал реализацию интерфейса `IOrderProcessor`, и добавить новый метод действия, который будет обрабатывать форму `POST` после того, когда пользователь нажмет кнопку `Complete order`. Листинг 9-17 показывает оба изменения.

**Листинг 9-17:** Завершаем класс CartController

```
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using SportsStore.WebUI.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace SportsStore.WebUI.Controllers
{
    public class CartController : Controller
    {
        private IProductRepository repository;
        private IOrderProcessor orderProcessor;
        public CartController(IProductRepository repo, IOrderProcessor proc)
        {
            repository = repo;
            orderProcessor = proc;
        }

        public ViewResult Index(Cart cart, string returnUrl)
        {
            return View(new CartIndexViewModel
            {
                Cart = cart,
                ReturnUrl = returnUrl
            });
        }

        public ViewResult Summary(Cart cart)
        {
            return View(cart);
        }

        [HttpPost]
        public ViewResult Checkout(Cart cart, ShippingDetails shippingDetails)
        {
            if (cart.Lines.Count() == 0)
            {
                ModelState.AddModelError("", "Sorry, your cart is empty!");
            }
            if (ModelState.IsValid)
            {
                orderProcessor.ProcessOrder(cart, shippingDetails);
                cart.Clear();
                return View("Completed");
            }
            else
            {
                return View(shippingDetails);
            }
        }

        public ViewResult Checkout()
        {
            return View(new ShippingDetails());
        }
        // ...other action methods omitted for brevity...
    }
}
```

Как видите, к методу действия `Checkout` теперь добавляется атрибут `HttpPost`, что означает, что он будет вызван для обработки запроса `POST` (в данном случае, когда пользователь отправляет форму). Опять же, мы полагаемся на механизм связывания данных как для параметра `ShippingDetails` (который создается автоматически на основе данных формы HTTP), так и для параметра `Cart` (который создается с помощью нашего пользовательского механизма связывания).

#### Примечание

*Из-за изменения конструктора мы должны обновить модульные тесты, которые мы создали для класса `CartController`. Тесты будут скомпилированы, если вы передадите `null` в новый параметра конструктора.*

MVC Framework проверяет ограничения валидации, которые мы применили к `ShippingDetails` помошью атрибутов `DataAnnotation` в листинге 9-17, и передает любые нарушения в наш метод действия через свойство `ModelState`. Мы можем увидеть, есть ли какие-нибудь проблемы, проверив свойство `ModelState.IsValid`. Обратите внимание, что мы вызываем метод `ModelState.AddModelError` для регистрации сообщения об ошибке, если нет товаров в корзине. Мы вкратце объясним, как отображать такие ошибки, и подробно разберем связывание данных и валидацию в главах 22 и 23.

### Модульный тест: Обработка заказов

Чтобы завершить модульное тестирование класса `CartController`, нам нужно протестировать поведение новой перегруженной версии метода `Checkout`. Хотя метод кажется коротким и простым, использование связывания данных MVC Framework означает, что происходит множество скрытых процессов, которые нужно проверить. Мы должны обработать заказ, только если в корзине есть товары и пользователь предоставил нам действительные реквизиты доставки. Во всех остальных случаях ему должно быть показано сообщение об ошибке. Вот первый тестовый метод:

```
[TestMethod]
public void Cannot_Checkout_Empty_Cart()
{
    // Arrange - create a mock order processor
    Mock<IOrderProcessor> mock = new Mock<IOrderProcessor>();

    // Arrange - create an empty cart
    Cart cart = new Cart();

    // Arrange - create shipping details
    ShippingDetails shippingDetails = new ShippingDetails();

    // Arrange - create an instance of the controller
    CartController target = new CartController(null, mock.Object);

    // Act
    ViewResult result = target.Checkout(cart, shippingDetails);

    // Assert - check that the order hasn't been passed on to the processor
    mock.Verify(m =>
        m.ProcessOrder(It.IsAny<Cart>(), It.IsAny<ShippingDetails>()), Times.Never());

    // Assert - check that the method is returning the default view
    Assert.AreEqual("", result.ViewName);

    // Assert - check that we are passing an invalid model to the view
    Assert.AreEqual(false, result.ViewData.ModelState.IsValid);
}
```

Этот тест гарантирует, что пользователь не сможет подтвердить покупку с пустой корзиной. Чтобы это проверить, мы утверждаем, что `ProcessOrder` имитированной реализации `IOrderProcessor` никогда не вызывается, что метод возвращает представление по умолчанию (которое снова отобразит данные, введенные пользователем, и даст возможность их исправить), и что состояние модели, которое передается в представление, отмечено как недопустимое. Может показаться, что наши утверждения дублируют друг друга, но нам нужны все три, чтобы гарантировать корректное поведение.

Следующий тестовый метод работает во многом таким же образом, но он внедряет сообщение об ошибке в модель представления и поручает механизму связывания данных сообщить об ошибке (что произойдет, когда клиент введет недействительные реквизиты доставки):

```
[TestMethod]
public void Cannot_Checkout_Invalid_ShippingDetails()
{
    // Arrange - create a mock order processor
    Mock<IOrderProcessor> mock = new Mock<IOrderProcessor>();

    // Arrange - create a cart with an item
    Cart cart = new Cart();
    cart.AddItem(new Product(), 1);

    // Arrange - create an instance of the controller
    CartController target = new CartController(null, mock.Object);

    // Arrange - add an error to the model
    target.ModelState.AddModelError("error", "error");

    // Act - try to checkout
    ViewResult result = target.Checkout(cart, new ShippingDetails());

    // Assert - check that the order hasn't been passed on to the processor
    mock.Verify(m =>
        m.ProcessOrder(It.IsAny<Cart>(), It.IsAny<ShippingDetails>()), Times.Never());

    // Assert - check that the method is returning the default view
    Assert.AreEqual("", result.ViewName);

    // Assert - check that we are passing an invalid model to the view
    Assert.AreEqual(false, result.ViewData.ModelState.IsValid);
}
```

Установив, что заказ не будет обрабатываться при пустой корзине или недействительных реквизитах, мы должны гарантировать, что обрабатываем заказы должным образом. Вот тест:

```
[TestMethod]
public void Can_Checkout_And_Submit_Order()
{
    // Arrange - create a mock order processor
    Mock<IOrderProcessor> mock = new Mock<IOrderProcessor>();

    // Arrange - create a cart with an item
    Cart cart = new Cart();
    cart.AddItem(new Product(), 1);

    // Arrange - create an instance of the controller
    CartController target = new CartController(null, mock.Object);

    // Act - try to checkout
    ViewResult result = target.Checkout(cart, new ShippingDetails());

    // Assert - check that the order has been passed on to the processor
}
```

```

mock.Verify(m =>
    m.ProcessOrder(It.IsAny<Cart>(), It.IsAny<ShippingDetails>()), Times.Once());

// Assert - check that the method is returning the Completed view
Assert.AreEqual("Completed", result.ViewName);

// Assert - check that we are passing a valid model to the view
Assert.AreEqual(true, result.ViewData.ModelState.IsValid);
}

```

Обратите внимание, что не нужно проверять, можем ли мы определить действительность реквизитов доставки. Это выполняется автоматически механизмом связывания с помощью атрибутов, которые мы применили к свойствам класса `ShippingDetails`.

## Отображаем ошибки валидации

Если пользователь введет недействительные реквизиты доставки, отдельные поля формы с проблемами будут подсвечены, но никаких сообщений отображаться не будет. Хуже того, если пользователь попытается подтвердить покупку с пустой корзиной, мы не дадим ему завершить заказ, но и не покажем никакого сообщения об ошибке. Чтобы решить эту проблему, мы должны добавить в представление `ValidationSummary`, как в главе 2. Листинг 9-18 показывает дополнение в представление `Checkout.cshtml`.

### Листинг 9-18: Добавляем ValidationSummary

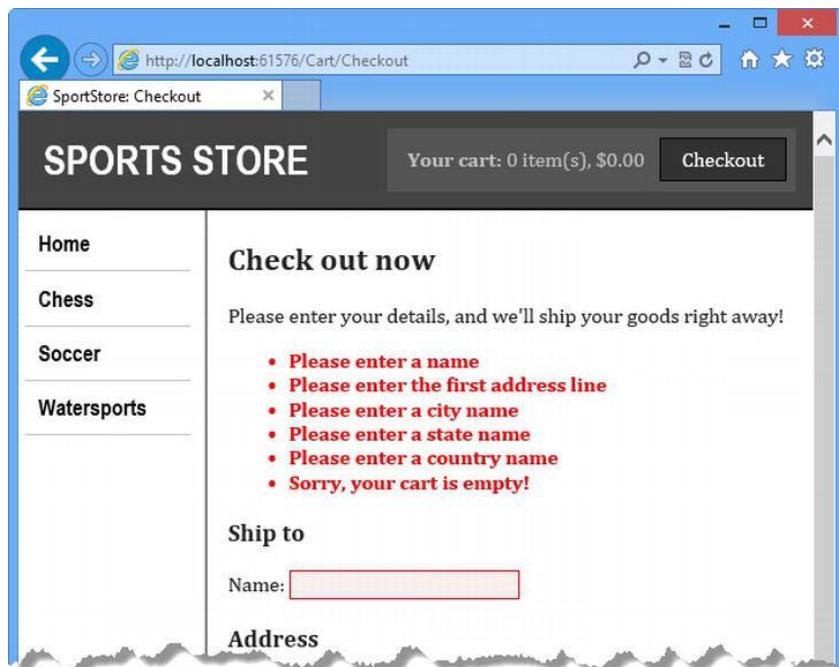
```

<h2>Check out now</h2>
Please enter your details, and we'll ship your goods right away!
@using (Html.BeginForm()) {
    @Html.ValidationSummary()
    <h3>Ship to</h3>
    <div>Name: @Html.EditorFor(x => x.Name)</div>

```

Теперь, когда пользователь введет недействительные реквизиты доставки или попытается подтвердить покупку с пустой корзиной, он увидит сообщение об ошибке, как показано на рисунке 9-7.

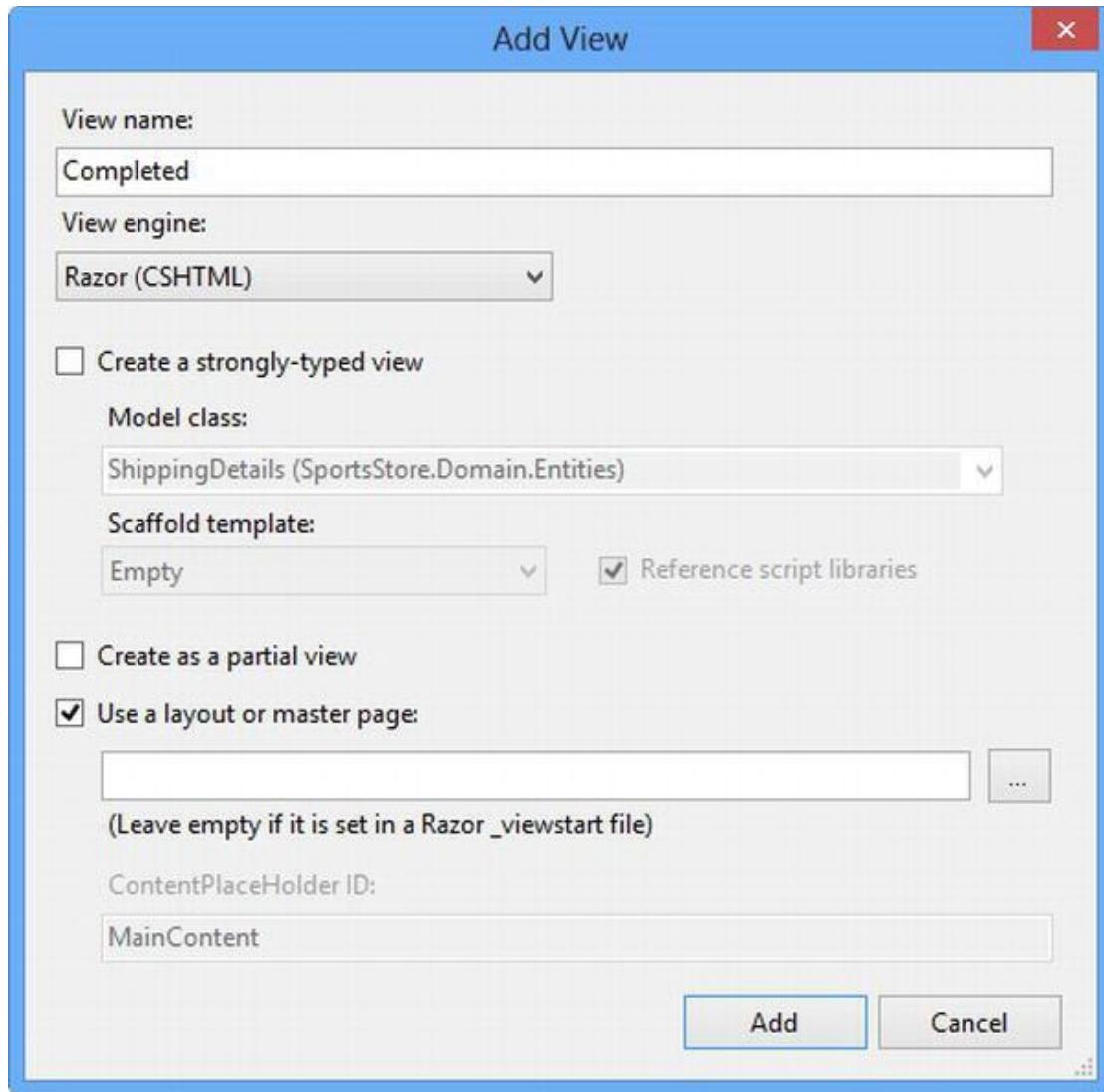
**Рисунок 9-7:** Отображение сообщений валидации



## Отображаем страницу подтверждения

Для завершения процесса подтверждения покупки мы покажем пользователю страницу с подтверждением, что заказ был обработан, и благодарностью за покупку. Кликните правой кнопкой мыши по любому методу действия в классе `CartController` и выберите `Add View` из контекстного меню. Назовите представление `Completed`, как показано на рисунке 9-8.

Рисунок 9-8: Создаем представление `Completed`



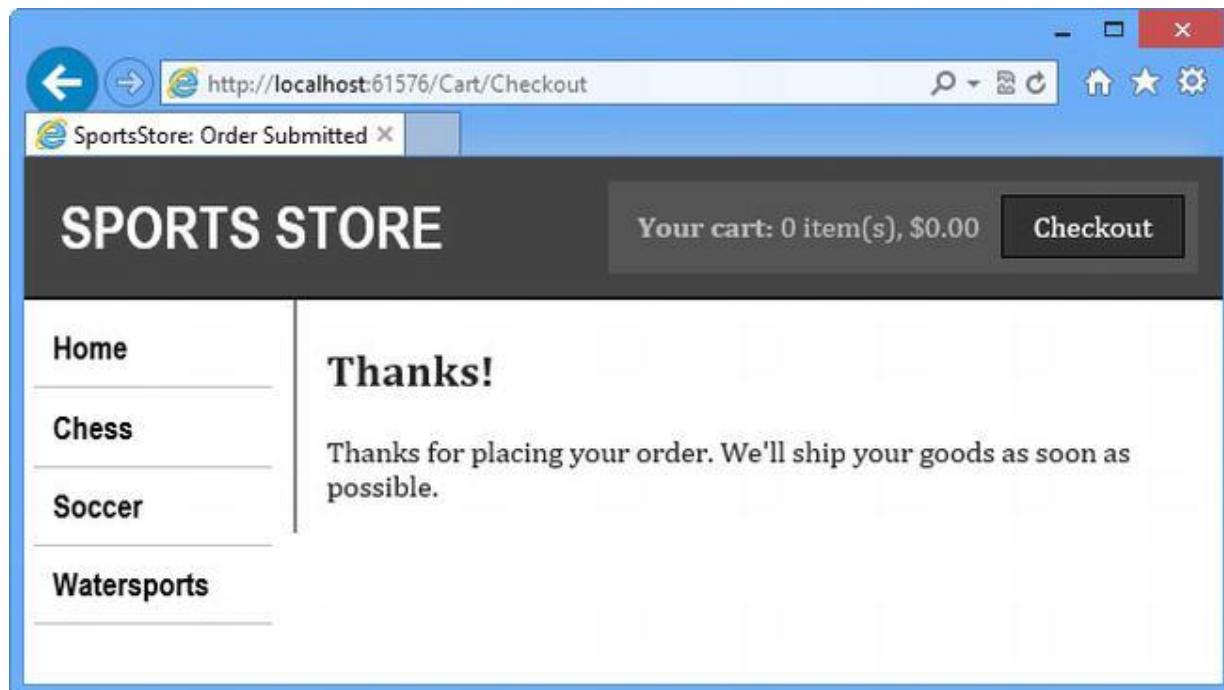
Мы не хотим, чтобы это представление было строго типизированным, потому что мы не собираемся передавать какие-либо модели представления между ним и контроллером. Мы хотим использовать макет, чтобы страница подтверждения выглядела соответственно остальным страницам приложения. Нажмите кнопку `Add`, чтобы создать представление, и отредактируйте его содержимое, чтобы оно соответствовало листингу 9-19.

Листинг 9-19: Представление `Completed.cshtml`

```
@{  
    ViewBag.Title = "SportsStore: Order Submitted";  
}  
  
<h2>Thanks!</h2>  
Thanks for placing your order. We'll ship your goods as soon as possible.
```

Теперь пользователь может пройти весь процесс, начиная с выбора товара и заканчивая подтверждением покупки. Если он предоставит действительные реквизиты доставки (и при наличии товаров в корзине), то, нажимая на кнопку `Complete order`, он попадет на страницу подтверждения, как показано на рисунке 9-9.

Рисунок 9-9: Страница с благодарностью



## Резюме

Мы завершили создание основной части приложения SportsStore, с которой работают пользователи. Может, мы и не станем конкурентами Amazon, но у нас есть каталог товаров, который можно просматривать по категориям и страницам, аккуратная корзина и простой процесс подтверждения покупки.

Хорошо разделенная архитектура означает, что мы можем легко изменять поведение любой части приложения, не беспокоясь о том, что это вызовет проблемы или противоречия в другом месте. Например, мы могли бы обрабатывать заказы, сохраняя их в базе данных, но это не повлияло бы на каталог товаров или любую другую часть приложения.

В следующей главе мы завершим приложение SportsStore, добавив средства администрации, которые позволят нам управлять каталогом и загружать, сохранять и отображать картинки для каждого товара.

# SportsStore: администрирование

В этой главе мы продолжим строить приложение SportsStore и создадим средства управления каталогом товаров для администратора сайта. Мы добавим поддержку создания, редактирования и удаления объектов из хранилища, а также загрузки изображений и их отображения вместе с описаниями товаров в каталоге.

## Управление каталогом

По соглашению для управления коллекциями товаров мы должны предоставить пользователю два типа страниц: страницу со списком и страницу редактирования, как показано на рисунке 10-1.

Рисунок 10-1: Эскиз пользовательского интерфейса CRUD для каталога товаров

На рисунке 10-1 представлены два эскиза пользовательского интерфейса для управления каталогом товаров:

- List Screen:** Страница списка товаров. Виджет имеет табличную структуру с двумя столбцами: "Item" и "Actions". Три элемента: Kayak, Lifejacket и Soccer ball, каждому соответствует пара ссылок "Edit | Delete". Внизу расположена кнопка "Add New Item".
- Edit Item: Kayak**: Страница редактирования товара Kayak. Имеет форму с четырьмя полями: Name (Kayak), Description (A boat for one pe...), Category (Watersports) и Price (\$): 275.00. Внизу находятся кнопки "Save" и "Cancel".

Вместе эти страницы позволяют пользователю создавать, читать, обновлять и удалять элементы в коллекции. В совокупности эти действия называются CRUD (Create-Read-Update-Delete – Создать-Прочитать-Обновить-Удалить). Разработчикам очень часто приходится реализовывать CRUD, так что Visual Studio предлагает генерировать MVC-контроллеры с готовыми методами действий для операций CRUD и шаблоны представлений, которые их поддерживают.

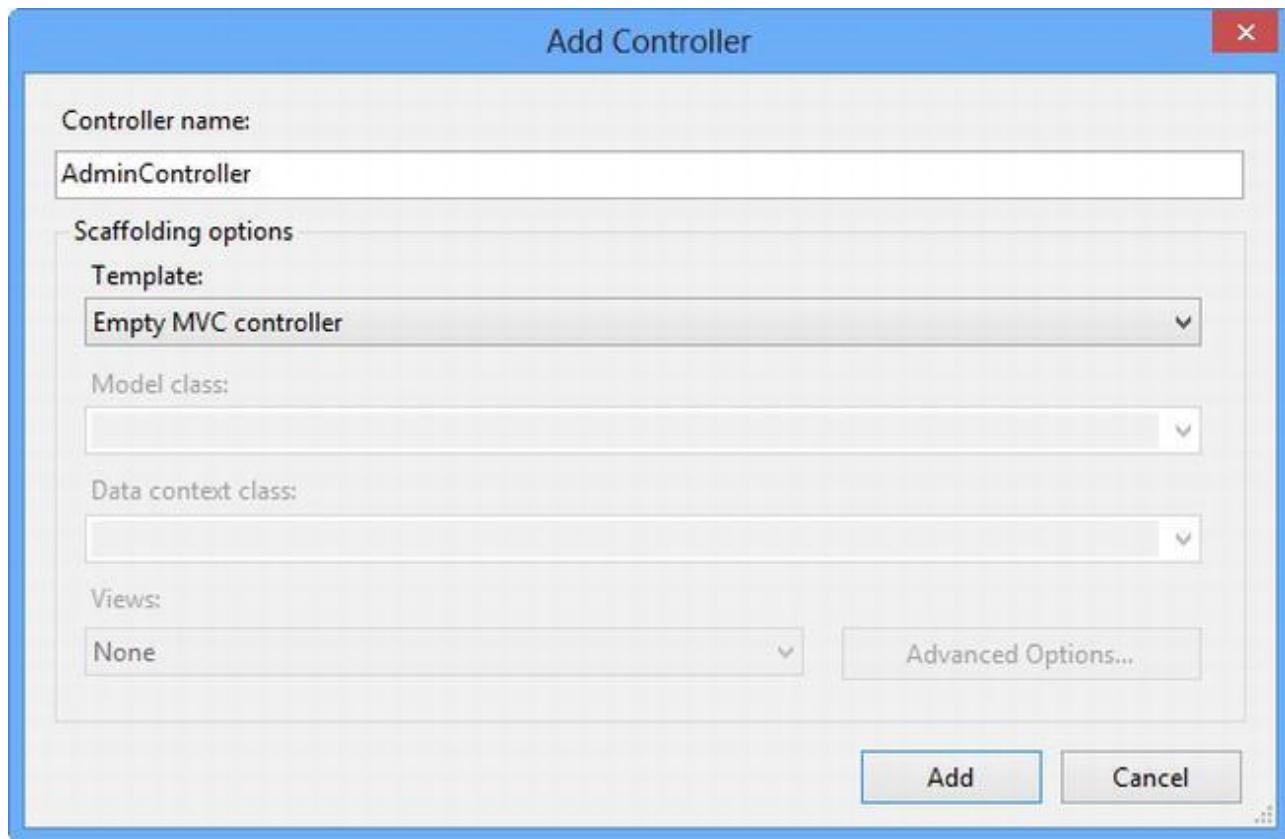
### Создаем контроллер CRUD

Мы создадим новый контроллер для обработки функций администрирования. Кликните правой кнопкой мыши папку `Controllers` в проекте `SportsStore.WebUI` и выберите `Add - Controller` из контекстного меню. Назовите контроллер `AdminController` и выберите из выпадающего списка `Template` пункт `Empty MVC Controller`, как показано на рисунке 10-2.

#### Примечание

*В Visual Studio есть несколько шаблонов для классов контроллеров, которые включают методы CRUD. Как мы уже говорили, они нам не нравятся и мы предпочитаем создавать классы контроллеров с нуля.*

**Рисунок 10-2:** Создание контроллера с помощью диалогового окна Add Controller



Нажмите кнопку Add, чтобы создать контроллер. Для поддержки страницы со списком, показанной на рисунке 10-1, нужно добавить метод действия, который будет отображать все товары в хранилище. Следуя соглашениям MVC Framework, мы назовем этот метод Index. Измените содержимое класса контроллера в соответствии с листингом 10-1.

**Листинг 10-1:** Метод действия Index

```
using SportsStore.Domain.Abstract;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace SportsStore.WebUI.Controllers
{
    public class AdminController : Controller
    {
        private IProductRepository repository;

        public AdminController(IProductRepository repo)
        {
            repository = repo;
        }

        public ViewResult Index()
        {
            return View(repository.Products);
        }
    }
}
```

## Модульный тест: Действие Index

Поведение метода `Index`, которое нас интересует, состоит в том, правильно ли он возвращает объекты `Product`, которые находятся в хранилище. Мы можем протестировать его, создав имитацию реализации хранилища и сравнив тестовые данные с данными, возвращенными методом действия. Вот модульный тест, который мы добавили в новый файл тестов под названием `AdminTests.cs`:

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using SportsStore.WebUI.Controllers;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.Mvc;

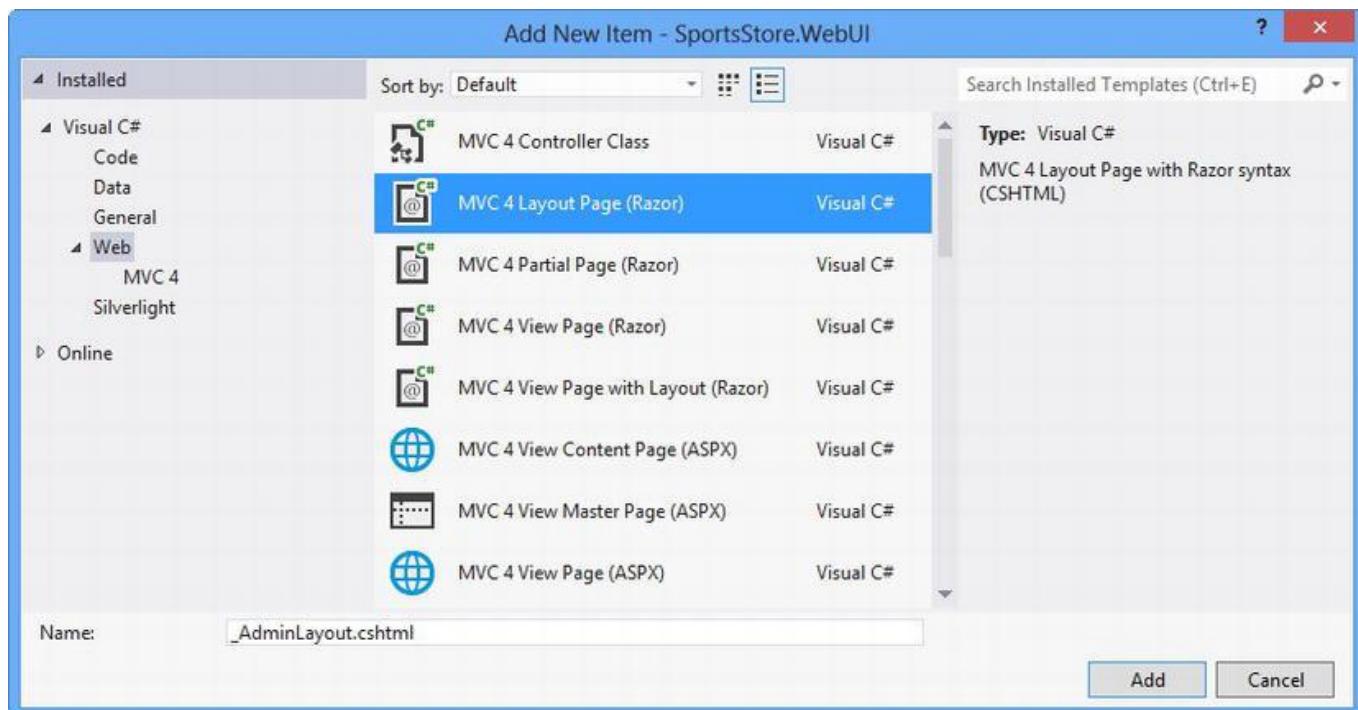
namespace SportsStore.UnitTests
{
    [TestClass]
    public class AdminTests
    {
        [TestMethod]
        public void Index_Contains_All_Products()
        {
            // Arrange - create the mock repository
            Mock<IProductRepository> mock = new Mock<IProductRepository>();
            mock.Setup(m => m.Products).Returns(new Product[] {
                new Product { ProductID = 1, Name = "P1" },
                new Product { ProductID = 2, Name = "P2" },
                new Product { ProductID = 3, Name = "P3" },
            }.AsQueryable());
            // Arrange - create a controller
            AdminController target = new AdminController(mock.Object);
            // Action
            Product[] result =
                ((IEnumerable<Product>)target.Index()).ViewData.Model.ToArray();
            // Assert
            Assert.AreEqual(result.Length, 3);
            Assert.AreEqual("P1", result[0].Name);
            Assert.AreEqual("P2", result[1].Name);
            Assert.AreEqual("P3", result[2].Name);
        }
    }
}
```

## Создаем новый макет

Мы собираемся создать новый макет для представлений администрирования `SportsStore`. Эта простая разметка будет представлять собой единственную точку, в которой мы сможем применить изменения ко всем представлениям администрирования.

Чтобы создать макет, щелкните правой кнопкой мыши папку `Views/Shared` в проекте `SportsStore.WebUI` и выберите `Add - New Item`. Выберите шаблон `MVC 4 Layout Page (Razor)` и назовите его `_AdminLayout.cshtml`, как показано на рисунке 10-3. Нажмите кнопку `Add`, чтобы создать новый файл.

**Рисунок 10-3:** Создаем новый макет Razor



Как мы уже объясняли ранее, по соглашению имя макета начинается с символа подчеркивания (\_). Razor также используется другой технологией Microsoft под названием WebMatrix, в которой символ подчеркивания нужен для того, чтобы предотвратить отправку страниц макетов в браузеры. В MVC такая защита не требуется, но соглашение об именах макетов так или иначе переносится на приложения MVC.

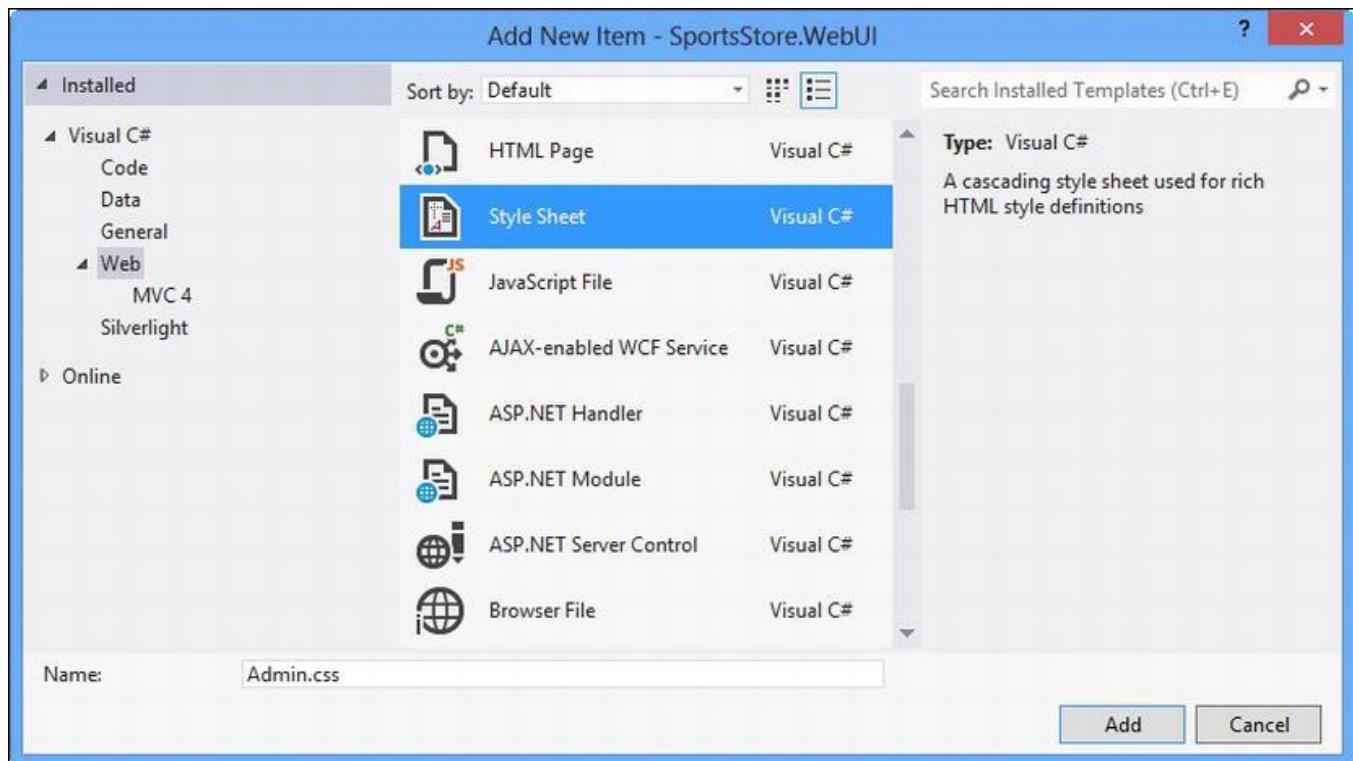
Мы хотим создать в макете ссылку на файл CSS, как показано в листинге 10-2.

**Листинг 10-2:** Файл \_AdminLayout.cshtml

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link href="~/Content/Admin.css" rel="stylesheet" type="text/css" />
    <title></title>
</head>
<body>
    <div>
        @RenderBody()
    </div>
</body>
</html>
```

Дополнение (выделено жирным шрифтом) является ссылкой на файл CSS под названием Admin.css в папке Content. Чтобы создать файл Admin.css, кликните правой кнопкой мыши папку Content, выберите пункт Add - New Item, шаблон Style Sheet и укажите имя Admin.css, как показано на рисунке 10-4.

**Рисунок 10-4:** Создание файла Admin.css



Замените содержимое файла Admin.css на стили, показанные в листинге 10-3.

### Листинг 10-3: CSS-стили для представлений администрирования

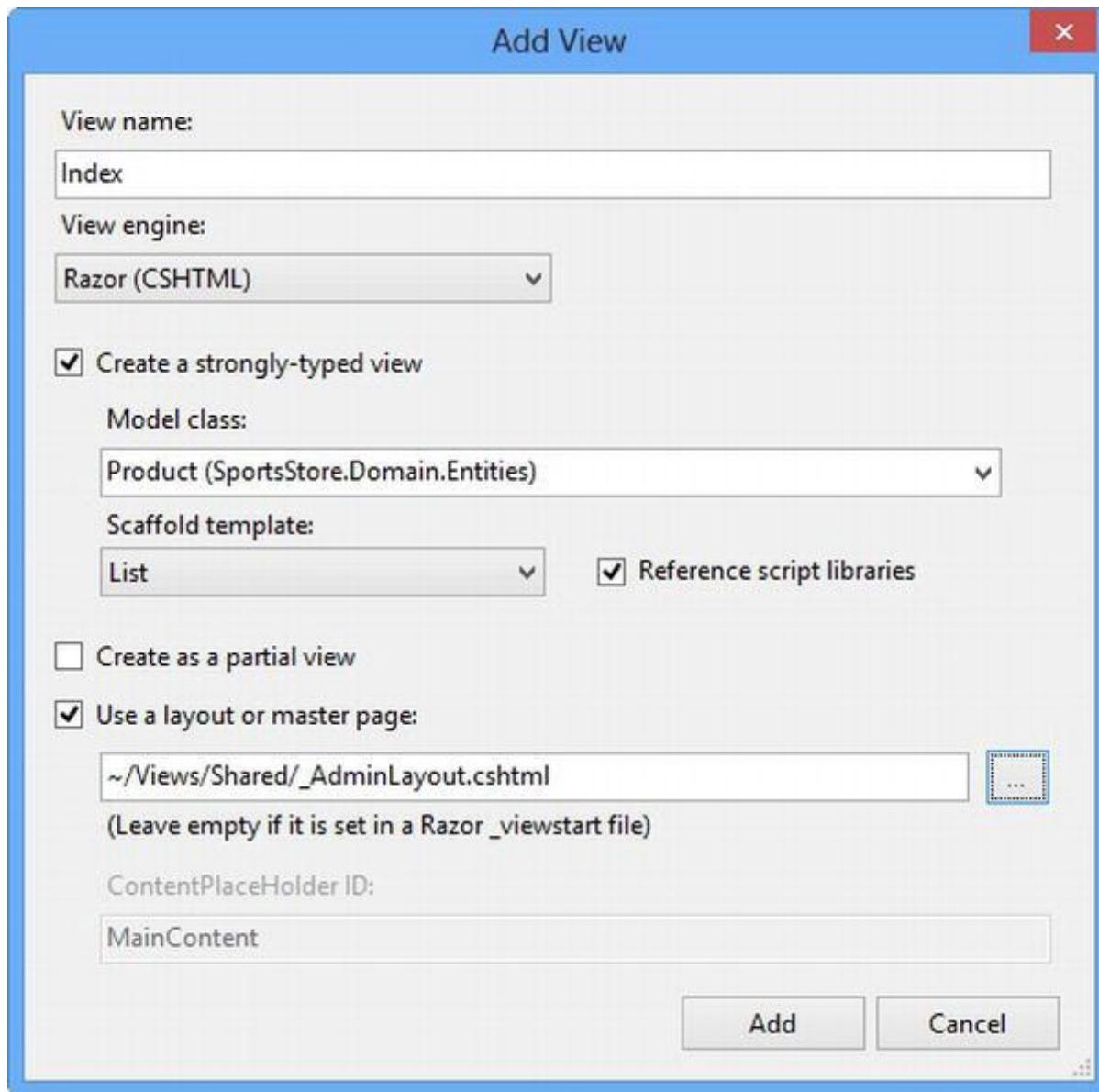
```
BODY, TD { font-family: Segoe UI, Verdana }
H1 { padding: .5em; padding-top: 0; font-weight: bold;
    font-size: 1.5em; border-bottom: 2px solid gray; }
DIV#content { padding: .9em; }
TABLE.Grid TD, TABLE.Grid TH { border-bottom: 1px dotted gray; text-align:left; }
TABLE.Grid { border-collapse: collapse; width:100%; }
TABLE.Grid TH.NumericCol, Table.Grid TD.NumericCol {
    text-align: right; padding-right: 1em; }
FORM {margin-bottom: 0px; }
DIV.Message { background: gray; color:white; padding: .2em; margin-top:.25em; }

.field-validation-error { color: red; display: block; }
.field-validation-valid { display: none; }
.input-validation-error { border: 1px solid red; background-color: #ffeeee; }
.validation-summary-errors { font-weight: bold; color: red; }
.validation-summary-valid { display: none; }
```

### Реализация представлений для страниц со списком

Теперь, когда новый макет готов, мы можем добавить в проект представление для метода действия Index контроллера Admin. Кликните правой кнопкой мыши по методу Index и выберите Add View из контекстного меню. Назовите представление Index, как показано на рисунке 10-5.

**Рисунок 10-5:** Создаем представление Index



Мы собираемся использовать заготовку (*scaffold view*) – то есть, представление, для которого Visual Studio сама создаст разметку в зависимости от того, какой мы выберем класс для строго типизированного представления. Чтобы ее создать, выберите `Product` из списка классов модели и шаблон заготовки `List`, как показано на рисунке 10-5.

### Примечание

Когда вы используете заготовку `List`, Visual Studio предполагает, что вы работаете с последовательностью `IEnumerable` типа модели представления, так что вы можете просто выбрать одиночную форму класса из списка.

Мы хотим применить наш вновь созданный макет, так что отметьте флагком опцию `Use a layout` и выберите файл `_AdminLayout.cshtml` из папки `Views/Shared`. Нажмите кнопку `Add`, чтобы создать представление. Заготовка, которую создаст Visual Studio, показана в листинге 10-4 (мы ее немножко подчистили, чтобы сделать более компактной и читабельной).

#### Листинг 10-4: Заготовка представления для страниц со списком

```
@model IEnumerable<SportsStore.Domain.Entities.Product>

{@{
    ViewBag.Title = "Index";
    Layout = "~/Views/Shared/_AdminLayout.cshtml";
}
<h2>Index</h2>
<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table>
    <tr>
        <th>@Html.DisplayNameFor(model => model.Name)</th>
        <th>@Html.DisplayNameFor(model => model.Description)</th>
        <th>@Html.DisplayNameFor(model => model.Price)</th>
        <th>@Html.DisplayNameFor(model => model.Category)</th>
        <th></th>
    </tr>
    @foreach (var item in Model) {
        <tr>
            <td>@Html.DisplayFor(modelItem => item.Name)</td>
            <td>@Html.DisplayFor(modelItem => item.Description)</td>
            <td>@Html.DisplayFor(modelItem => item.Price)</td>
            <td>@Html.DisplayFor(modelItem => item.Category)</td>
            <td>
                @Html.ActionLink("Edit", "Edit", new { id = item.ProductID }) |
                @Html.ActionLink("Details", "Details", new { id = item.ProductID }) |
                @Html.ActionLink("Delete", "Delete", new { id = item.ProductID })
            </td>
        </tr>
    }
</table>
```

Visual Studio смотрит на тип объекта модели представления и генерирует в таблице элементы, которые соответствуют его свойствам. Вы можете увидеть, как визуализируется это представление, если запустите приложение и перейдете по ссылке /Admin/Index. Результаты показаны на рисунке 10-6.

Рисунок 10-6: Визуализация представления для страниц со списком

Name	Description	Price	Category	
Kayak	A boat for one person	275.00	Watersports	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Lifejacket	Protective and fashionable	48.95	Watersports	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Soccer Ball	FIFA-approved size and weight	19.50	Soccer	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Corner Flags	Give your playing field a professional touch	34.95	Soccer	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Stadium	Flat-packed 35,000-seat stadium	79500.00	Soccer	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Thinking Cap	Improve your brain efficiency by 75%	16.00	Chess	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Unsteady Chair	Secretly give your opponent a disadvantage	29.95	Chess	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Human Chess Board	A fun game for the family	75.00	Chess	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Bling-Bling King	Gold-plated, diamond-studded King	1200.00	Chess	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>

Заготовка во многом помогает нам с настройками. У нас имеются столбцы для каждого из свойств класса `Product` и ссылки на другие операции CRUD, которые ведут к методам действий того же контроллера. Но стоит отметить, что она содержит излишнюю разметку. Кроме того, мы хотим использовать в представлении CSS, который мы создали ранее. Отредактируйте файл `Index.cshtml` в соответствии с листингом 10-5.

**Листинг 10-5:** Изменяем представление `Index.cshtml`

```
@model IEnumerable<SportsStore.Domain.Entities.Product>

{@{
    ViewBag.Title = "Admin: All Products";
    Layout = "~/Views/Shared/_AdminLayout.cshtml";
}

<h1>All Products</h1>
<table class="Grid">
    <tr>
        <th>ID</th>
        <th>Name</th>
        <th class="NumericCol">Price</th>
        <th>Actions</th>
    </tr>
    @foreach (var item in Model) {
        <tr>
            <td>@item.ProductID</td>
            <td>@Html.ActionLink(item.Name, "Edit", new { item.ProductID })</td>
            <td class="NumericCol">@item.Price.ToString("c")</td>
            <td>
                @using (Html.BeginForm("Delete", "Admin"))
                {
                    @Html.Hidden("ProductID", item.ProductID)
                    <input type="submit" value="Delete" />
                }
            </td>
        </tr>
    }
</table>
<p>@Html.ActionLink("Add a new product", "Create")</p>
```

Это представление визуализирует информацию более компактно, оно опускает некоторые свойства из класса `Product` и по-другому представляет ссылки на конкретные товары. Вы можете увидеть, как оно выглядит, на рисунке 10-7.

**Рисунок 10-7:** Визуализация измененного представления `Index`

ID	Name	Price	Actions
1	Kayak	\$275.00	Delete
2	Lifejacket	\$48.95	Delete
3	Soccer Ball	\$19.50	Delete
4	Corner Flags	\$34.95	Delete
5	Stadium	\$79,500.00	Delete
6	Thinking Cap	\$16.00	Delete
7	Unsteady Chair	\$29.95	Delete
9	Human Chess Board	\$75.00	Delete
10	Bling-Bling King	\$1,200.00	Delete

Add a new product

Теперь у нас готова страница со списком. Администратор может просматривать товары в каталоге, а также использовать ссылки или кнопки для добавления, удаления и просмотра элементов. В следующих разделах мы добавим функциональность для поддержки каждого из этих действий.

## Редактирование товаров

Чтобы обеспечить поддержку создания и обновления, мы добавим страницу редактирования товаров, как показано на рисунке 10-1. Эту работу мы выполним в два этапа:

- Отобразим страницу, которая позволит администратору изменять значения свойств товара.
- Добавим метод действия, который будет обрабатывать изменения после их отправки.

### Создаем метод действия Edit

В листинге 10-6 показан метод `Edit`, который мы добавили к классу `AdminController`. Это метод действия, который мы указали в вызовах к вспомогательному методу действия `Html.ActionLink` в представлении `Index`.

#### Листинг 10-6: Метод Edit

```
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace SportsStore.WebUI.Controllers
{
    public class AdminController : Controller
    {
        private IProductRepository repository;

        public AdminController(IProductRepository repo)
        {
            repository = repo;
        }

        public ViewResult Index()
        {
            return View(repository.Products);
        }

        public ViewResult Edit(int productId)
        {
            Product product = repository.Products
                .FirstOrDefault(p => p.ProductID == productId);
            return View(product);
        }
    }
}
```

Этот простой метод находит товар с `ID`, который соответствует параметру `productId`, и передает его в качестве объекта модели представления.

## Модульный тест: метод действия Edit

Мы хотим протестировать два вида поведения в методе действия Edit. Первое состоит в том, получаем ли мы правильный товар, когда предоставляем действительное значение ID. Очевидно, мы хотим убедиться, что отредактируем именно тот товар, который собирались. Второй вид поведения заключается в том, что мы не получаем никакого товара вообще, когда мы предоставляем недействительное значение ID. Вот тестовые методы, которые мы добавили в файл AdminTests.cs:

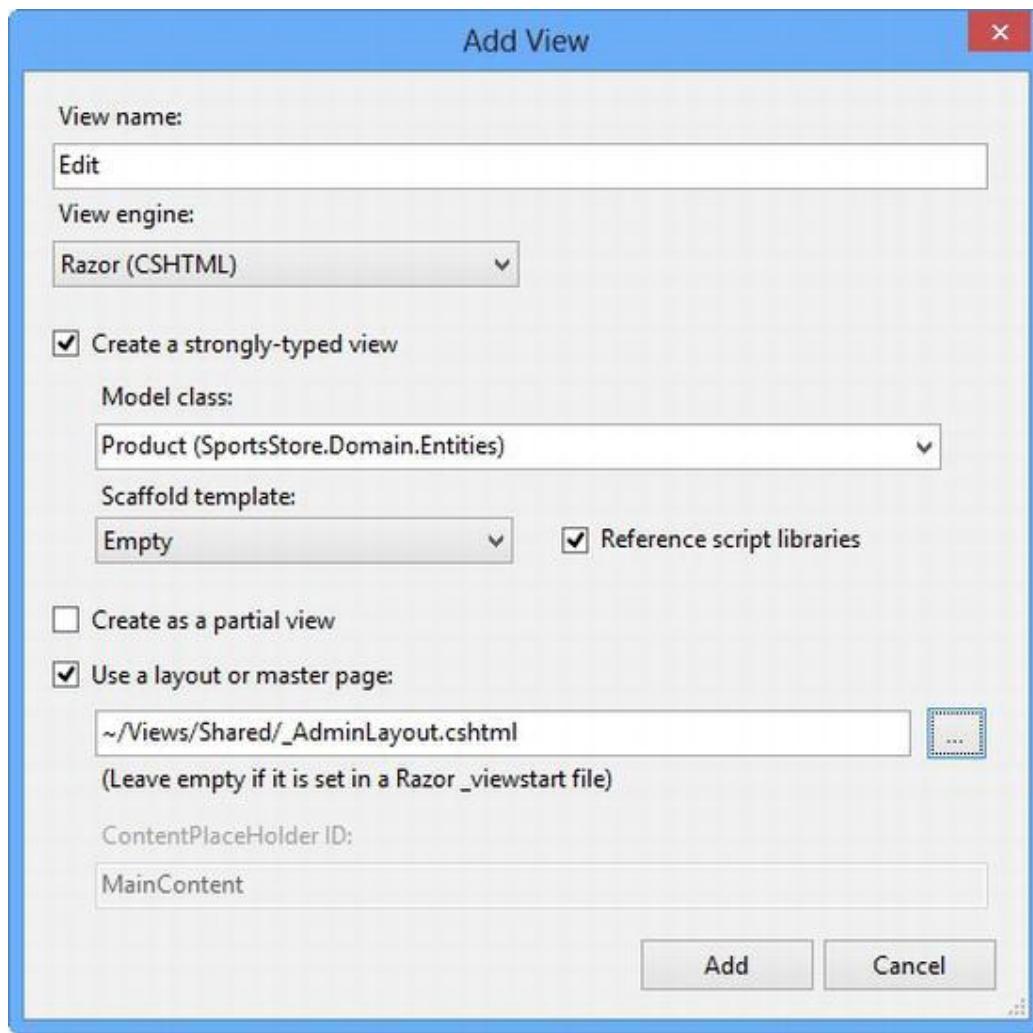
```
[TestMethod]
public void Can_Edit_Product()
{
    // Arrange - create the mock repository
    Mock<IProductRepository> mock = new Mock<IProductRepository>();
    mock.Setup(m => m.Products).Returns(new Product[] {
        new Product {ProductID = 1, Name = "P1"},
        new Product {ProductID = 2, Name = "P2"},
        new Product {ProductID = 3, Name = "P3"},
    }.AsQueryable());
    // Arrange - create the controller
    AdminController target = new AdminController(mock.Object);
    // Act
    Product p1 = target.Edit(1).ViewData.Model as Product;
    Product p2 = target.Edit(2).ViewData.Model as Product;
    Product p3 = target.Edit(3).ViewData.Model as Product;
    // Assert
    Assert.AreEqual(1, p1.ProductID);
    Assert.AreEqual(2, p2.ProductID);
    Assert.AreEqual(3, p3.ProductID);
}

[TestMethod]
public void Cannot_Edit_Nonexistent_Product()
{
    // Arrange - create the mock repository
    Mock<IProductRepository> mock = new Mock<IProductRepository>();
    mock.Setup(m => m.Products).Returns(new Product[] {
        new Product {ProductID = 1, Name = "P1"},
        new Product {ProductID = 2, Name = "P2"},
        new Product {ProductID = 3, Name = "P3"},
    }.AsQueryable());
    // Arrange - create the controller
    AdminController target = new AdminController(mock.Object);
    // Act
    Product result = (Product)target.Edit(4).ViewData.Model;
    // Assert
    Assert.IsNull(result);
}
```

## Создаем представление редактирования

Теперь, когда у нас готов метод действия, мы можем создать представление, которое он будет визуализировать. Кликните правой кнопкой мыши по методу действия Edit и выберите пункт Add View. Оставьте имя представления Edit, отметьте флагжком опцию Create a strongly-typed view и убедитесь, что в качестве класса модели выбран класс Product, как показано на рисунке 10-8.

**Рисунок 10-8:** Создаем представление редактирования



Для CRUD-операции `Edit` имеется заготовка, которую вы можете выбрать, чтобы посмотреть, что создает Visual Studio. Мы же снова будем использовать свою разметку, поэтому выбрали из списка опций для заготовок `Empty`. Не забудьте отметить флажком опцию `Use a layout` и в качестве представления выберите `_AdminLayout.cshtml`. Нажмите кнопку `Add`, чтобы создать представление, которое будет размещено в папке `Views/Admin`. Изменить его содержимое в соответствии с листингом 10-7.

#### Листинг 10-7: Представление `Edit`

```
@model SportsStore.Domain.Entities.Product

@{
    ViewBag.Title = "Admin: Edit " + @Model.Name;
    Layout = "~/Views/Shared/_AdminLayout.cshtml";
}

<h1>Edit @Model.Name</h1>
@using (Html.BeginForm())
{
    @Html.EditorForModel()
    <input type="submit" value="Save" />
    @Html.ActionLink("Cancel and return to List", "Index")
}
```

Чтобы не писать разметку для каждой метки и поля ввода вручную, мы вызвали вспомогательный метод `Html.EditorForModel`. Этот метод сообщает MVC Framework создать интерфейс редактирования, для чего она проверит тип модели, в данном случае класс `Product`.

Чтобы увидеть страницу, которая создается представлением `Edit`, запустите приложение и перейдите по ссылке `/Admin/Index`. Кликните по одному из товаров, и вы увидите страницу, показанную на рисунке 10-9.

**Рисунок 10-9:** Страница, сгенерированная с помощью вспомогательного метода `EditorForModel`

The screenshot shows a web browser window with a blue header bar. The address bar displays `http://localhost:61576`. The main content area has a title **Edit Corner Flags**. Below the title is a form with the following fields:

- ProductID**: Input field containing **4**.
- Name**: Input field containing **Corner Flags**.
- Description**: Input field containing **Give your playing field a ...**.
- Price**: Input field containing **34.95**.
- Category**: Input field containing **Soccer**.

At the bottom of the form are two buttons: **Save** and **Cancel and return to List**.

Давайте будем честными - метод `EditorForModel` удобен, но дает не самые привлекательные результаты. К тому же, мы не хотим, чтобы администратор мог видеть или редактировать атрибут `ProductID`, а текстовое поле для свойства `Description` слишком мало.

Мы можем дать MVC Framework указания касательно того, как создавать редакторы для свойств, с помощью метаданных модели. Это позволит нам применять атрибуты к свойствам нового класса модели, которые повлияют на вывод метода `Html.EditorForModel`. В листинге 10-8 показано, как использовать метаданные в классе `Product` в проекте `SportsStore.Domain`.

### Листинг 10-8: Используем метаданные модели

```
using System.ComponentModel.DataAnnotations;
using System.Web.Mvc;

namespace SportsStore.Domain.Entities
{
    public class Product
    {
        [HiddenInput(DisplayValue = false)]
        public int ProductID { get; set; }

        public string Name { get; set; }

        [DataType(DataType.MultilineText)]
        public string Description { get; set; }

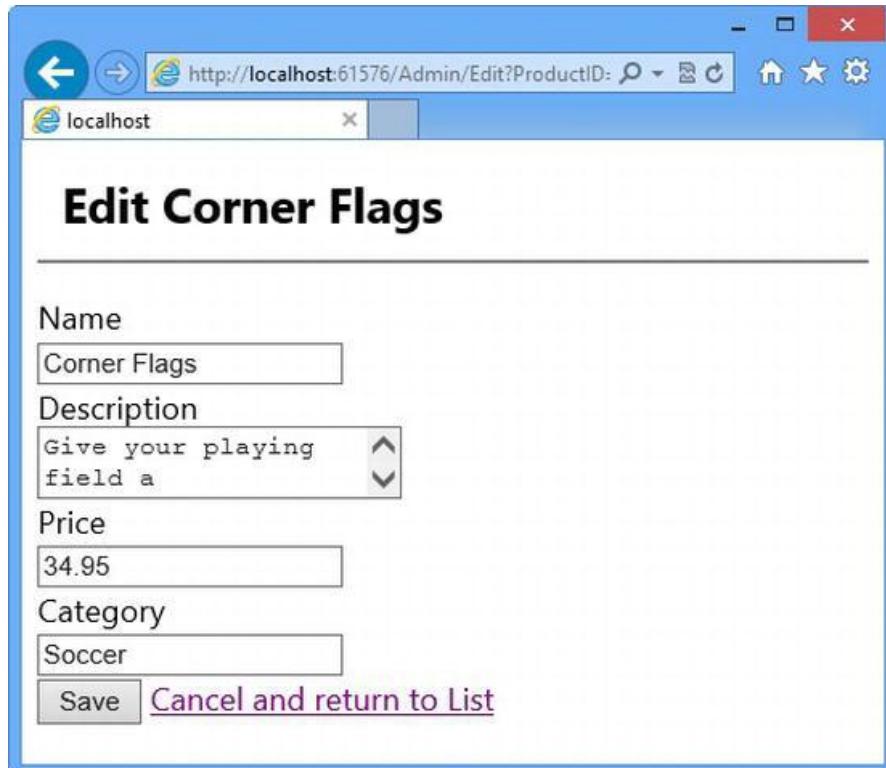
        public decimal Price { get; set; }

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

Атрибут `HiddenInput` сообщает MVC Framework, что свойство нужно визуализировать как скрытый элемент формы, а атрибут `DataType` позволяет указать, как значение должно отображаться и редактироваться. В данном случае мы выбрали опцию `MultilineText`. Атрибут `HiddenInput` является частью пространства имен `System.Web.Mvc`, и атрибут `DataType` - частью пространства имен `System.ComponentModel.DataAnnotations`, что объясняет, почему нам нужно было добавить ссылки на коллекции этих имен в проект `SportsStore.Domain` в главе 7.

Рисунок 10-10 показывает страницу `Edit` после применения метаданных. Вы больше не можете видеть или редактировать свойство `ProductId`, и у вас есть многострочное текстовое поле для ввода описания. Однако UI по-прежнему выглядит довольно скромно.

Рисунок 10-10: Эффект применения метаданных



Мы можем несколько улучшить страницу с помощью CSS. Когда MVC Framework создает поле ввода для каждого свойства, она присваивает им различные классы CSS. Если вы посмотрите на исходный код страницы, показанной на рисунке 10-10, вы увидите, что элементу текстового поля для описания товара был присвоен CSS-класс `text-box-multi-line`:

```
<textarea class="text-box multi-line" id="Description" name="Description">  
    Give your playing field a professional touch  
</textarea>
```

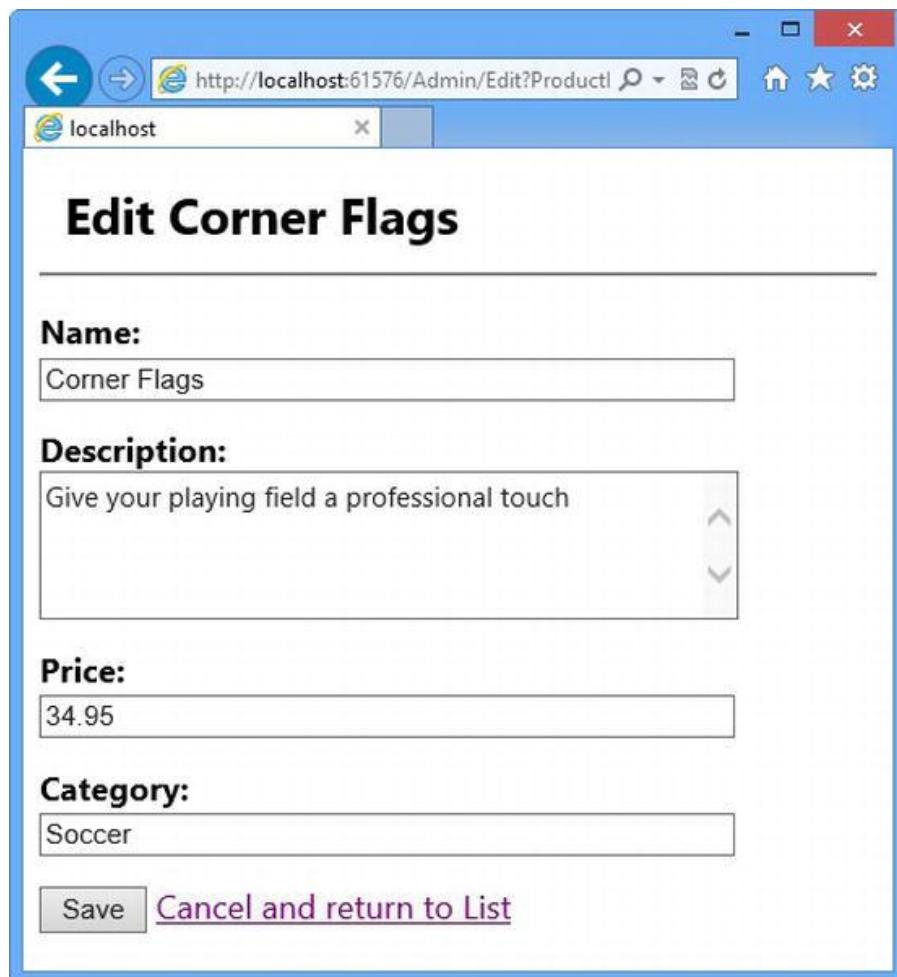
Другим классам присваиваются подобные элементы HTML, и мы можем улучшить внешний вид представления Edit, добавив стили из листинга 10-9 в файл `Admin.css` из папки Content проекта `SportsStore.WebUI`. Эти стили назначаются разным классам, которые добавляются к элементам HTML вспомогательным методом `EditorForModel`.

#### Листинг 10-9: CSS-стили для элементов редактирования

```
.editor-field { margin-bottom: .8em; }  
.editor-label { font-weight: bold; }  
.editor-label:after { content: ":" }  
.text-box { width: 25em; }  
.multi-line { height: 5em; font-family: Segoe UI, Verdana; }
```

Рисунок 10-11 показывает эффект применения этих стилей в представлении Edit. Визуализированное представление выглядит все еще довольно незамысловато, но оно функционально и удовлетворяет нашим запросам.

Рисунок 10-11: Применение CSS к элементам редактирования



Как вы видели в этом примере, страница, которую создает шаблонный вспомогательный метод вроде `EditorForModel`, не будет всегда отвечать вашим требованиям. Мы обсудим использование и настройку шаблонных вспомогательных методов подробнее в главе 20.

## Обновляем хранилище Product

Прежде чем мы сможем обрабатывать изменения, нам нужно обновить хранилище `Product` так, чтобы можно было их сохранять. Во-первых, мы добавим в интерфейс `IProductRepository` новый метод, который показан в листинге 10-10. (Напомним, что этот интерфейс можно найти в папке `Abstract` проекта `SportsStore.Domain`).

### Листинг 10-10: Добавляем метод в интерфейс хранилища

```
using System.Linq;
using SportsStore.Domain.Entities;
namespace SportsStore.Domain.Abstract
{
    public interface IProductRepository
    {
        IQueryable<Product> Products { get; }
        void SaveProduct(Product product);
    }
}
```

Затем мы можем добавить этот метод в реализацию хранилища Entity Framework, в класс `Concrete/EFProductRepository`, как показано в листинге 10-11.

### Листинг 10-11: Реализация метода `SaveProduct`

```
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using System.Linq;
namespace SportsStore.Domain.Concrete
{
    public class EFProductRepository : IProductRepository
    {
        private EFDbContext context = new EFDbContext();

        public IQueryable<Product> Products
        {
            get { return context.Products; }
        }

        public void SaveProduct(Product product)
        {
            if (product.ProductID == 0)
            {
                context.Products.Add(product);
            }
            else
            {
                Product dbEntry = context.Products.Find(product.ProductID);
                if (dbEntry != null)
                {
                    dbEntry.Name = product.Name;
                    dbEntry.Description = product.Description;
                    dbEntry.Price = product.Price;
                    dbEntry.Category = product.Category;
                }
            }
            context.SaveChanges();
        }
    }
}
```

}

Реализация метода `SaveChanges` добавляет товар в хранилище, если `ProductID` равен 0; в противном случае она применяет изменения к существующей записи в базе данных.

Мы не хотим подробно разбирать Entity Framework, потому что, как мы объясняли ранее, это очень обширная тема, но метод `SaveProduct` имеет некоторое отношение к дизайну приложения MVC.

Мы знаем, что необходимо выполнить обновление, когда получаем параметр `Product` с `ProductID`, не равным нулю. Для этого мы получаем из хранилища объект `Product` с таким же `ProductID` и обновляем каждое свойство так, чтобы оно совпадало с объектом параметра.

Это необходимо потому, что Entity Framework отслеживает объекты, которые он создает из базы данных. Объект, который передается в метод `SaveChanges`, создается MVC Framework с помощью стандартной модели связывания, что означает, что Entity Framework ничего не узнает об объекте параметра и не применит обновления к базе данных. Есть много путей решения этой проблемы, и мы выбрали самый простой: размещение соответствующего объекта, о котором будет знать Entity Framework, и явное его обновление.

Альтернативный подход заключается в создании пользовательского механизма связывания, который будет только получать объекты из хранилища. Этот подход может показаться более гладким, но для него потребовалось бы добавить возможность поиска в интерфейс хранилища, чтобы мы могли бы найти объекты `Product` по значению `ProductID`.

Недостаток такого подхода заключается в том, что мы начали бы добавлять функциональность к абстрактному определению хранилища для того, чтобы обойти ограничения конкретной реализации. Если мы будем переключаться между реализациями хранилища в будущем, есть риск того, что придется реализовать возможность поиска, для которой нет готовой поддержки в новой технологии хранения. Гибкость MVC Framework, как, к примеру, метод `SaveProduct`, может показаться хорошей возможностью, чтобы не использовать обходные пути, но она поставит под удар дизайн вашего приложения.

## Обработка запросов Edit POST

На данный момент мы готовы реализовать перегруженный метод действия `Edit` в контроллере `Admin`, который будет обрабатывать запросы `POST`, которые отправляются нажатием кнопки `Save`. Новый метод показан в листинге 10-12.

**Листинг 10-12:** Добавляем метод действия `Edit` для обработки запросов `POST`

```
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
namespace SportsStore.WebUI.Controllers
{
    public class AdminController : Controller
    {
        private IProductRepository repository;
        public AdminController(IProductRepository repo)
        {
            repository = repo;
        }
        public ViewResult Index()
```

```

    {
        return View(repository.Products);
    }
    public ViewResult Edit(int productId)
    {
        Product product = repository.Products
            .FirstOrDefault(p => p.ProductID == productId);
        return View(product);
    }
    [HttpPost]
    public ActionResult Edit(Product product)
    {
        if (ModelState.IsValid)
        {
            repository.SaveProduct(product);
            TempData["message"] = string.Format("{0} has been saved", product.Name);
            return RedirectToAction("Index");
        }
        else
        {
            // there is something wrong with the data values
            return View(product);
        }
    }
}
}

```

Мы убеждаемся, что механизм связывания провел валидацию предоставленных пользователем данных, прочитав значение свойства `ModelState.IsValid`. Если все в порядке, мы сохраняем изменения в хранилище, а затем вызываем метод действия `Index`, чтобы вернуть пользователя к списку товаров. Если есть проблема с данными, мы снова визуализируем представление `Edit`, чтобы пользователь мог внести исправления.

После сохранения изменений в хранилище, мы сохраняем сообщение с помощью объекта `TempData`. Этот набор пар ключ/значение похож на данные сессии и `ViewBag`, которые мы использовали ранее. Его основное отличие от данных сессии заключается в том, что `TempData` удаляется в конце запроса HTTP.

Обратите внимание, что мы возвращаем тип `ActionResult` из метода `Edit`. До сих пор мы использовали тип `ViewResult`. `ViewResult` наследует от `ActionResult` и используется в тех случаях, когда платформа должна визуализировать представление. Тем не менее, доступны другие типы `ActionResult`, и один из них возвращается методом `RedirectToAction`. Мы используем его в методе действия `Edit` для вызова метода действия `Index`.

В этой ситуации мы не можем использовать `ViewBag`, так как нам нужно перенаправить пользователя. `ViewBag` передает данные между контроллером и представлением, и он не может хранить данные дольше, чем длится текущий запрос HTTP. Мы могли бы использовать данные сессии, но тогда сообщение будет храниться, пока мы явно его не удалим, а нам не хочется этого делать. Таким образом, `TempData` нам идеально подходит. Данные ограничиваются сессией одного пользователя (так что пользователи не видят другие `TempData`) и будут сохранены до тех пор, пока мы их не прочитаем. Они понадобятся нам в представлении, визуализированном тем методом действия, к которому мы перенаправили пользователя.

### **Модульный тест: получение данных от метода Edit**

Для обработки запросов POST метода действия `Edit` мы должны убедиться, что для сохранения в хранилище передаются только действительные обновления объекта `Product`, созданного механизмом

связывания. Мы также хотим гарантировать, что недействительные обновления, в которых существует ошибка модели, не передаются в хранилище. Вот тестовые методы:

```
[TestMethod]
public void Can_Save_Valid_Changes()
{
    // Arrange - create mock repository
    Mock<IProductRepository> mock = new Mock<IProductRepository>();
    // Arrange - create the controller
    AdminController target = new AdminController(mock.Object);
    // Arrange - create a product
    Product product = new Product { Name = "Test" };
    // Act - try to save the product
    ActionResult result = target.Edit(product);
    // Assert - check that the repository was called
    mock.Verify(m => m.SaveProduct(product));
    // Assert - check the method result type
    Assert.IsInstanceOfType(result, typeof(ViewResult));
}

[TestMethod]
public void Cannot_Save_Invalid_Changes()
{
    // Arrange - create mock repository
    Mock<IProductRepository> mock = new Mock<IProductRepository>();
    // Arrange - create the controller
    AdminController target = new AdminController(mock.Object);
    // Arrange - create a product
    Product product = new Product { Name = "Test" };
    // Arrange - add an error to the model state
    target.ModelState.AddModelError("error", "error");
    // Act - try to save the product
    ActionResult result = target.Edit(product);
    // Assert - check that the repository was not called
    mock.Verify(m => m.SaveProduct(It.IsAny<Product>()), Times.Never());
    // Assert - check the method result type
    Assert.IsInstanceOfType(result, typeof(ViewResult));
}
```

## Отображаем сообщение с подтверждением

Мы будем отображать сохраненное с помощью TempData сообщение в файле макета \_AdminLayout.cshtml. Обрабатывая сообщение в шаблоне, мы сможем создавать сообщения в любом представлении, которое использует этот шаблон, не создавая для них дополнительные блоки Razor. В листинге 10-13 показаны изменения в файле.

**Листинг 10-13:** Обработка сообщения ViewBag в макете

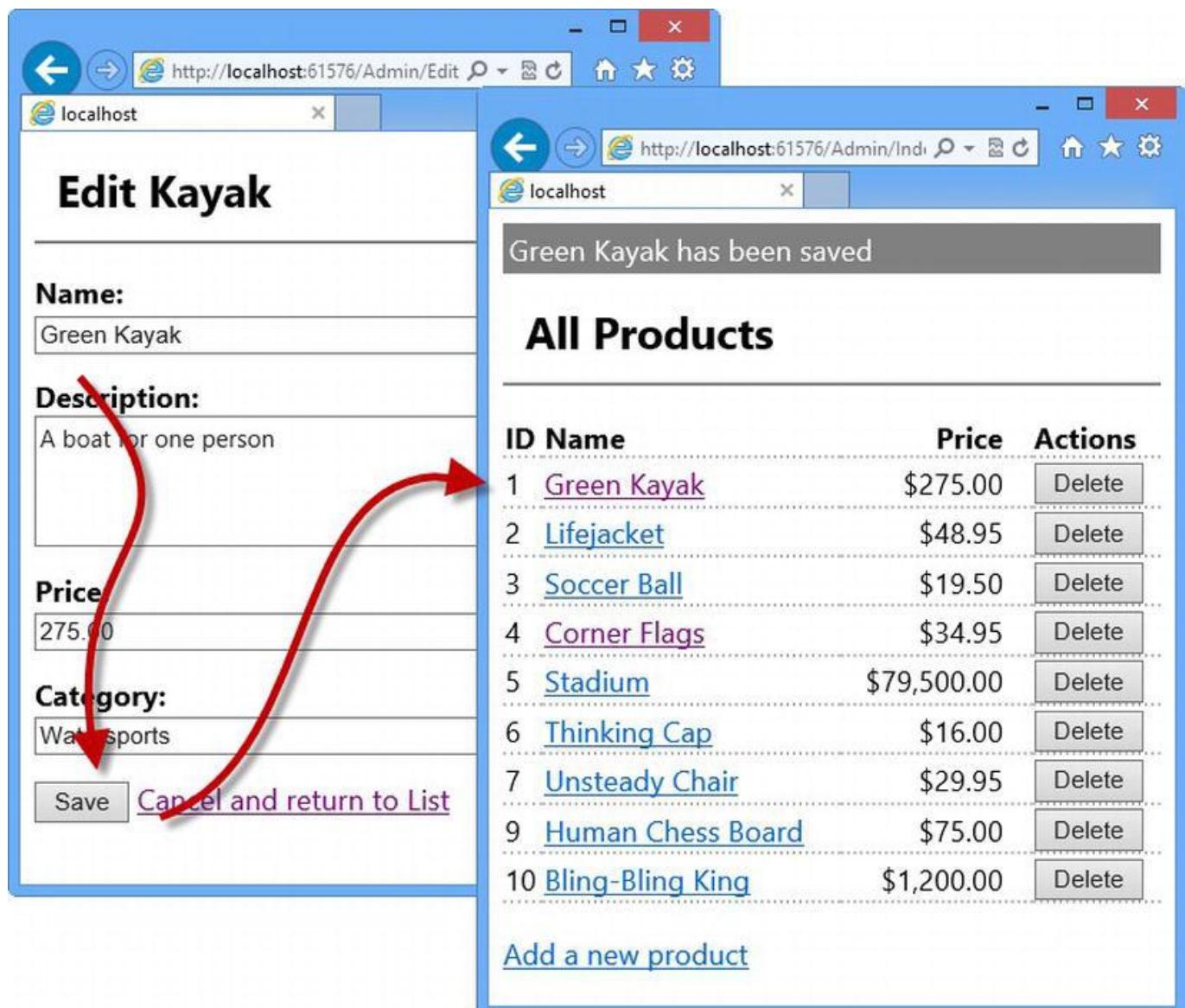
```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link href="~/Content/Admin.css" rel="stylesheet" type="text/css" />
    <title></title>
</head>
<body>
    <div>
        @if (TempData["message"] != null) {
            <div class="Message">@TempData["message"]</div>
        }
        @RenderBody()
    </div>
</body>
</html>
```

### Примечание

Преимущество обработки сообщения в шаблоне заключается в том, что пользователи увидят его не зависимо от того, какая страница будет отображаться после того, как они сохранят изменения. На данный момент мы возвращаем их на страницу со списком товаров, но мы могли бы изменить рабочий поток и отображать какое-нибудь другое представление, а пользователи все равно увидят сообщение (при условии, что другое представление использует тот же макет).

Теперь у нас готовы все элементы, которые нужны для тестирования функции редактирования товаров. Запустите приложение, перейдите по ссылке Admin/Index, и внесите какие-нибудь изменения. Нажмите кнопку Save. Вы вернетесь к представлению со списком, в котором будет отображаться сообщение TempData, как показано на рисунке 10-12.

Рисунок 10-12: Редактируем товар и видим сообщение TempData



Сообщение исчезнет, если вы перезагрузите страницу со списком товаров, потому что TempData удаляется после прочтения. Это очень удобно, так как мы не хотим, чтобы сохранялись старые сообщения.

## Добавляем валидацию модели

В таких случаях нам всегда нужно добавлять правила валидации для нашей сущности модели. На данный момент администратор может отправить отрицательные цены или пустое описание, и SportsStore запросто сохранит их в базе данных. В листинге 10-14 показано применение атрибута DataAnnotations к классу Product, аналогичное применению атрибута к классу ShippingDetails в предыдущей главе.

**Листинг 10-14:** Применение атрибутов валидации к классу Product

```
using System.ComponentModel.DataAnnotations;
using System.Web.Mvc;
namespace SportsStore.Domain.Entities
{
    public class Product
    {
        [HiddenInput(DisplayValue = false)]
        public int ProductID { get; set; }

        [Required(ErrorMessage = "Please enter a product name")]
        public string Name { get; set; }

        [DataType(DataType.MultilineText)]
        [Required(ErrorMessage = "Please enter a description")]
        public string Description { get; set; }

        [Required]
        [Range(0.01, double.MaxValue, ErrorMessage = "Please enter a positive price")]
        public decimal Price { get; set; }

        [Required(ErrorMessage = "Please specify a category")]
        public string Category { get; set; }
    }
}
```

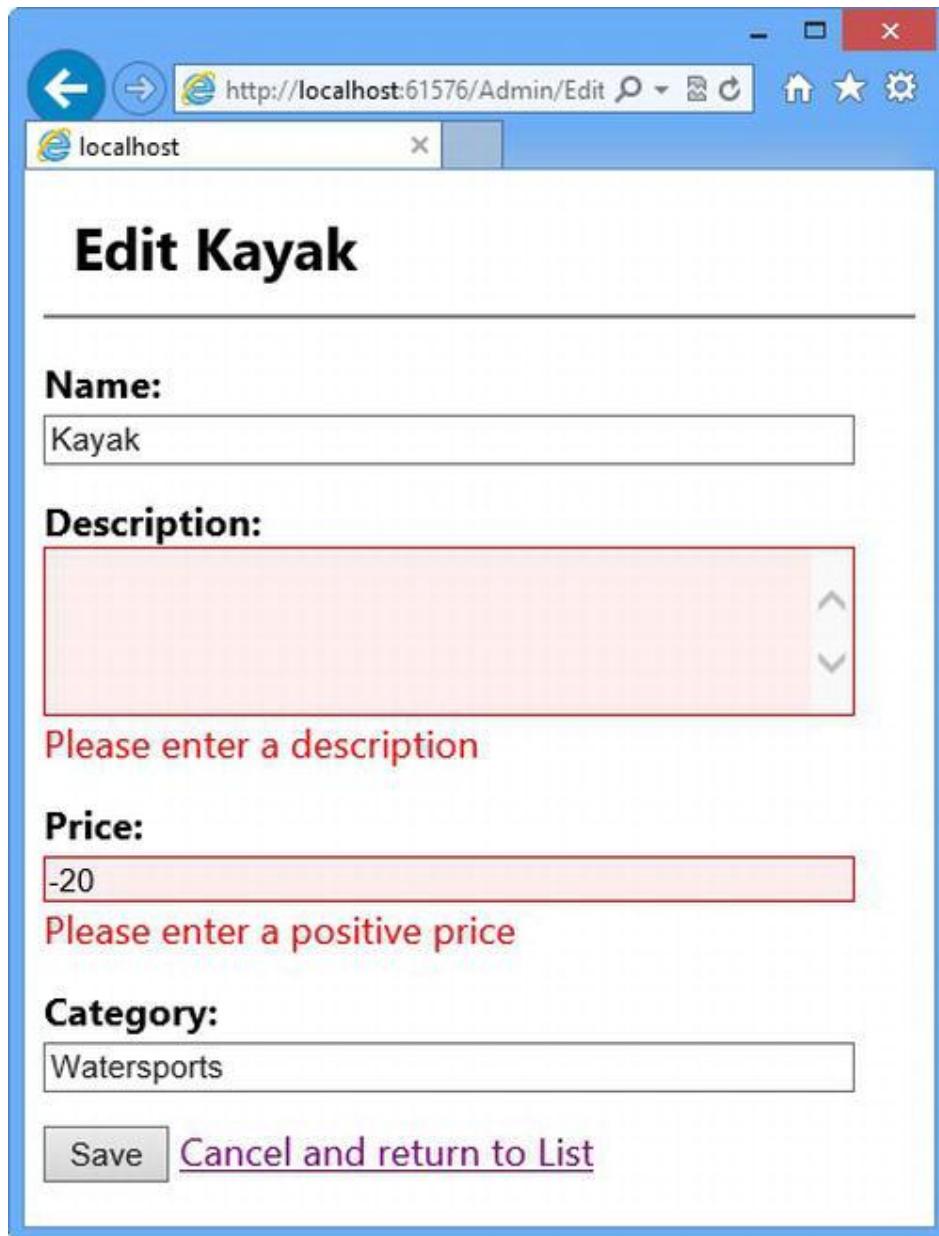
### Примечание

Теперь в классе Product больше атрибутов, чем свойств. Не волнуйтесь, если вам кажется, что они сделают класс нечитаемым. Эти атрибуты можно переместить в другой класс и сообщить MVC, где их найти. Мы покажем, как это сделать, в главе 23.

---

Когда мы использовали вспомогательный метод Html.EditorForModel для создания элементов формы для редактирования объекта Product, MVC Framework добавляла разметку и применяла классы CSS, необходимые для отображения ошибок валидации рядом с формой. На рисунке 10-13 показано, как при редактировании товара проявляется нарушение правил валидации, которые мы применили к классу Product.

Рисунок 10-13: Валидация данных при редактировании товаров



### Включаем валидацию на стороне клиента

На данный момент валидация данных применяется только тогда, когда администратор отправляет поправки на сервер. Но, если есть проблемы с введенными данными, необходимо предоставить пользователям немедленную обратную связь. Поэтому разработчикам часто требуется выполнять валидацию на стороне клиента, то есть проверку данных браузером с помощью JavaScript. MVC Framework может выполнять валидацию на стороне клиента на основе атрибутов DataAnnotations, которые мы применили к классу доменной модели.

Эта функция включена по умолчанию, но до сих пор она не работала, потому что мы не добавили ссылки на необходимые библиотеки JavaScript. Проще всего добавить эти ссылки в файл `_AdminLayout.cshtml`, чтобы валидация на стороне клиента могла работать на любой странице, которая использует этот макет. В листинге 10-15 показаны изменения в макете. Функция валидации на стороне клиента MVC основана на JavaScript-библиотеке JQuery, что понятно из имен файлов сценария.

### Листинг 10-15: Импорт файлов JavaScript для валидации на стороне клиента

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link href="~/Content/Admin.css" rel="stylesheet" type="text/css" />
    <script src="~/Scripts/jquery-1.7.1.js"></script>
    <script src="~/Scripts/jquery.validate.min.js"></script>
    <script src="~/Scripts/jquery.validate.unobtrusive.min.js"></script>
    <title></title>
</head>
<body>
    <div>
        @if ( TempData["message"] != null) {
            <div class="Message">@TempData["message"]</div>
        }
        @RenderBody()
    </div>
</body>
</html>
```

С этими дополнениями валидация на стороне клиента будет работать для всех представлений администрирования. Пользователь будет видеть те же самые сообщения об ошибках, потому что при валидации на стороне клиента используются те же CSS-классы, которые используются при серверной валидации, но ответ будет появляться без задержек и без отправки запроса на сервер. В большинстве случаев валидация на стороне клиента - полезная функция, но если по какой-то причине вы не хотите ее использовать, необходимо использовать следующие операторы:

```
HtmlHelper.ClientValidationEnabled = false;
HtmlHelper.UnobtrusiveJavaScriptEnabled = false;
```

Если вы поместите их в представление или контроллер, то валидация на стороне клиента будет отключена только для текущего действия. Вы можете отключить валидацию на стороне клиента для всего приложения, используя эти операторы в методе Application\_Start Global.asax или добавив в файл Web.config следующие значения:

```
<configuration>
    <appSettings>
        <add key="ClientValidationEnabled" value="false" />
        <add key="UnobtrusiveJavaScriptEnabled" value="false" />
    </appSettings>
</configuration>
```

## Создаем новые товары

Далее мы реализуем метод действия Create, на который ведет ссылка Add a new product на странице со списком товаров. Он позволит администратору добавлять новые элементы в каталог товаров. Чтобы добавить поддержку создания новых товаров, потребуется только одно небольшое дополнение и одно изменение в нашем приложении. Это отличный пример возможностей и гибкости хорошо продуманного приложения MVC. Во-первых, добавьте метод Create, показанный в листинге 10-16, в класс AdminController.

### Листинг 10-16: Добавляем метод действия Create в AdminController

```
public ViewResult Create() {
    return View("Edit", new Product());
}
```

Метод `Create` не визуализирует свое представление по умолчанию. Вместо этого он указывает, что должно быть использовано представление `Edit`. То, что один метод действия использует представление, обычно связанное с другим представлением, является вполне приемлемым. В данном случае мы внедряем новый объект `Product` в качестве модели представления, так что представление `Edit` заполняется пустыми полями.

Это приводит нас к следующей модификации. Обычно мы ожидаем, что форма отправляет запрос к действию, которое ее визуализировало, и именно это по умолчанию предполагает `Html.BeginForm`, когда генерирует HTML-форму. Тем не менее, для нашего метода `Create` это не работает, потому что мы хотим, чтобы форма отправляла запрос к действию `Edit`, чтобы можно было сохранить добавленные данные о товаре. Чтобы это исправить, можно использовать перегруженную версию вспомогательного метода `Html.BeginForm`, чтобы указать, что целью формы, сгенерированной в представлении `Edit`, является метод действия `Edit` контроллера `Admin`. Листинг 10-17 содержит изменение, которое мы внесли в файл представления `Views/Admin/Edit.cshtml`.

#### Листинг 10-17: Явно указываем метод действия и контроллер для формы

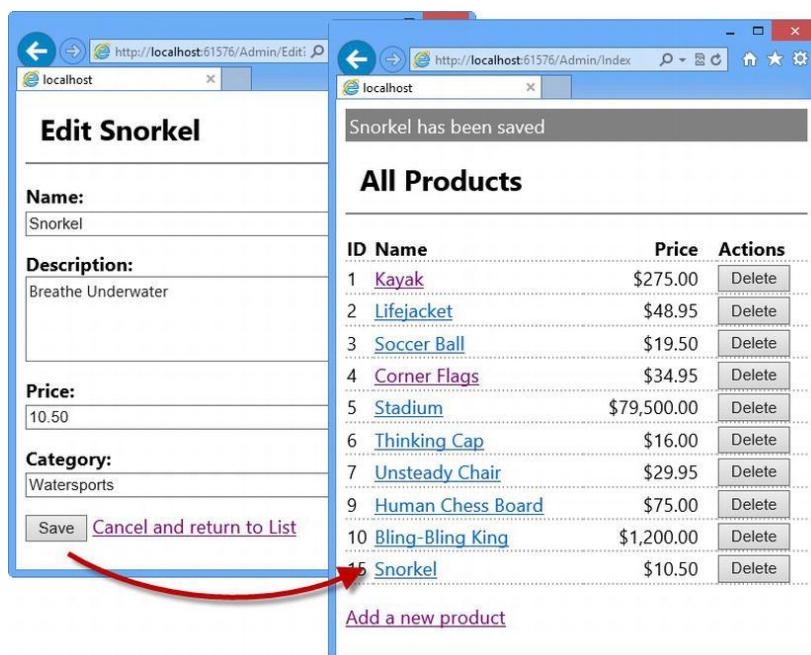
```
@model SportsStore.Domain.Entities.Product
{
    ViewBag.Title = "Admin: Edit " + @Model.Name;
    Layout = "~/Views/Shared/_AdminLayout.cshtml";
}

<h1>Edit @Model.Name</h1>

@using (Html.BeginForm("Edit", "Admin"))
{
    @Html.EditorForModel()
    <input type="submit" value="Save" />
    @Html.ActionLink("Cancel and return to List", "Index")
}
```

Теперь форма всегда будет отправлена действию `Edit` независимо от того, каким действием она была визуализирована. Мы можем создавать товары, перейдя по ссылке `Add a new product` и заполнив поля, как показано на рисунке 10-14.

**Рисунок 10-14 : Добавляем новый товар в каталог**



## Удаляем товары

Добавить поддержку для удаления товаров довольно просто. Для начала мы добавим новый метод в интерфейс `IProductRepository`, как показано в листинге 10-18.

### Листинг 10-18: Добавляем метод для удаления товаров

```
using System.Linq;
using SportsStore.Domain.Entities;
namespace SportsStore.Domain.Abstract
{
    public interface IProductRepository
    {
        IQueryable<Product> Products { get; }
        void SaveProduct(Product product);
        Product DeleteProduct(int productID);
    }
}
```

Далее мы реализуем этот метод в классе хранилища Entity Framework, `EFProductRepository`, как показано в листинге 10-19.

### Листинг 10-19: Реализуем поддержку удаления в классе хранилища Entity Framework

```
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using System.Linq;
namespace SportsStore.Domain.Concrete
{
    public class EFProductRepository : IProductRepository
    {
        private EFDbContext context = new EFDbContext();
        public IQueryable<Product> Products
        {
            get { return context.Products; }
        }
        public void SaveProduct(Product product)
        {
            if (product.ProductID == 0)
            {
                context.Products.Add(product);
            }
            else
            {
                Product dbEntry = context.Products.Find(product.ProductID);
                if (dbEntry != null)
                {
                    dbEntry.Name = product.Name;
                    dbEntry.Description = product.Description;
                    dbEntry.Price = product.Price;
                    dbEntry.Category = product.Category;
                }
            }
            context.SaveChanges();
        }
        public Product DeleteProduct(int productID)
        {
            Product dbEntry = context.Products.Find(productID);
            if (dbEntry != null)
            {
                context.Products.Remove(dbEntry);
                context.SaveChanges();
            }
            return dbEntry;
        }
    }
}
```

```
    }
}
}
```

Последним шагом будет реализация метода действия `Delete` в контроллере `Admin`. Этот метод действия будет поддерживать только запросы `POST`, так как удаление объектов не является идемпотентной операцией. Как мы объясним в главе 14, браузеры и кэши могут делать запросы `GET` без явного согласия пользователя, поэтому мы должны проявить осторожность, чтобы избежать внесения изменений в результат запросов `GET`. Листинг 10-20 содержит новый метод действия.

#### Листинг 10-20: Метод действия `Delete`

```
[HttpPost]
public ActionResult Delete(int productId) {
    Product deletedProduct = repository.DeleteProduct(productId);
    if (deletedProduct != null) {
        TempData["message"] = string.Format("{0} was deleted", deletedProduct.Name);
    }
    return RedirectToAction("Index");
}
```

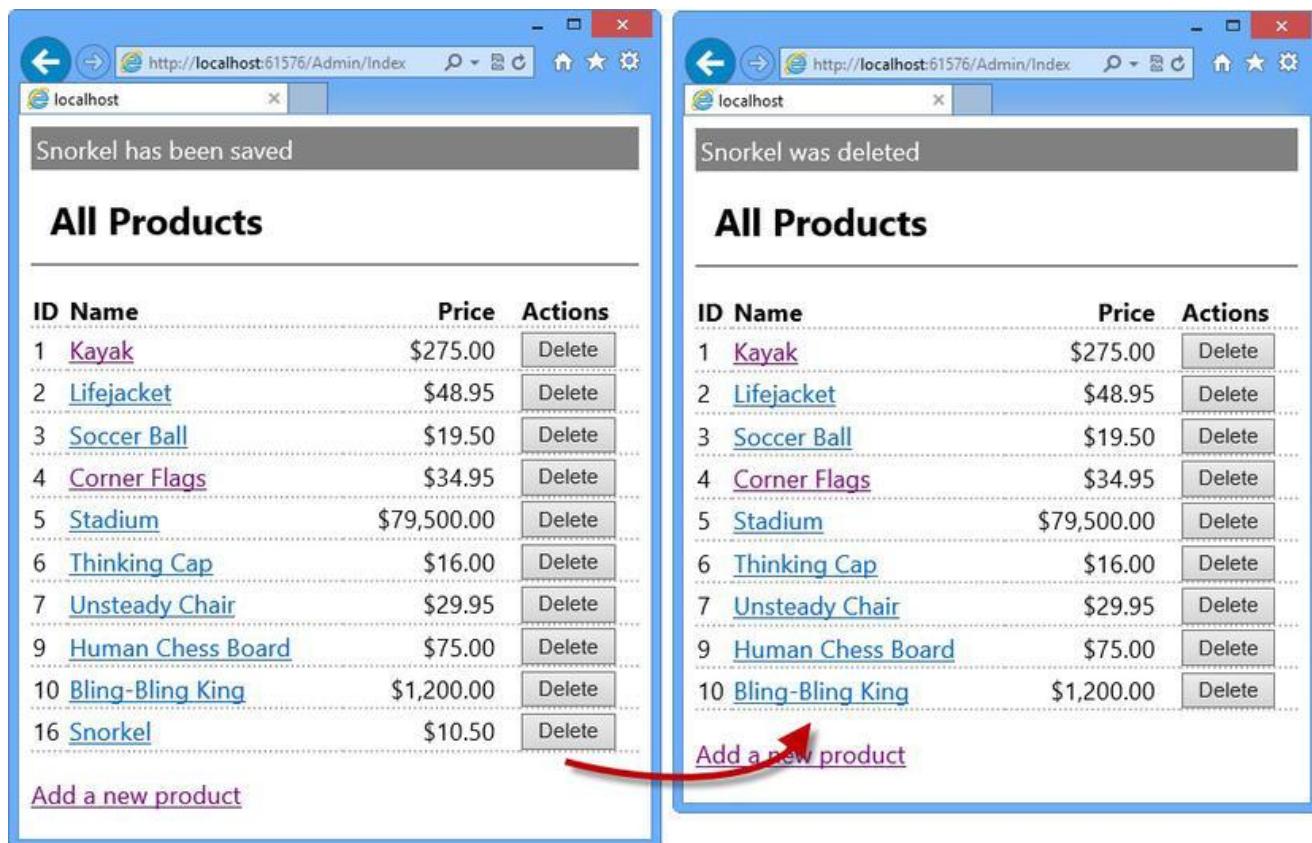
#### Модульный тест: удаление товаров

Мы хотим протестировать базовое поведение метода действия `Delete`: когда действительный `ProductID` передается ему в качестве параметра, он должен вызвать метод хранилища `DeleteProduct` и передать значение `ProductID`, которое должно быть удалено. Вот тест:

```
[TestMethod]
public void Can_Delete_Valid_Products()
{
    // Arrange - create a Product
    Product prod = new Product { ProductID = 2, Name = "Test" };
    // Arrange - create the mock repository
    Mock<IProductRepository> mock = new Mock<IProductRepository>();
    mock.Setup(m => m.Products).Returns(new Product[] {
        new Product { ProductID = 1, Name = "P1" },
        prod,
        new Product { ProductID = 3, Name = "P3" },
    }).AsQueryable();
    // Arrange - create the controller
    AdminController target = new AdminController(mock.Object);
    // Act - delete the product
    target.Delete(prod.ProductID);
    // Assert - ensure that the repository delete method was
    // called with the correct Product
    mock.Verify(m => m.DeleteProduct(prod.ProductID));
}
```

Вы сможете увидеть работу новой функции, просто нажав одну из кнопок `Delete` на странице со списком товаров, как показано на рисунке 10-15. Как видите, мы воспользовались переменной `TempData` для отображения сообщения о том, что продукт будет удален из каталога.

Рисунок 10-15: Удаляем товар из каталога



## Резюме

В этой главе мы создали поддержку администрирования и показали вам, как реализовать операции CRUD, которые позволяют администратору создавать, читать, обновлять и удалять товары из хранилища.

В следующей главе мы покажем вам, как обеспечить защиту функций администрирования, чтобы они не были доступны для всех пользователей, и добавим последние штрихи для завершения функциональности SportsStore.

# SportsStore: безопасность и последние штрихи

В предыдущей главе мы добавили поддержку администрирования приложения SportsStore, и от вашего внимания не должен был ускользнуть тот факт, что любой пользователь сможет изменить каталог товаров, если мы развернем приложение прямо сейчас. Для этого нужно только знать, что функции администрирования доступны по ссылке `Admin/Index`. В этой главе мы покажем вам, как предотвратить использование средств администрирования случайными пользователями, защитив доступ к контроллеру `Admin` с помощью пароля.

Когда безопасность будет реализована, мы завершим приложение SportsStore, добавив поддержку изображения товаров. Это может показаться простой функцией, но она требует применения некоторых интересных техник MVC.

## Безопасность административной части

Поскольку ASP.NET MVC построена на ядре платформы ASP.NET, у нас есть доступ к forms-аутентификации, которая представляет собой универсальную систему для отслеживания входа в приложение. В нашем примере мы просто покажем вам, как провести базовую настройку.

Если вы откроете файл `Web.config` (в корневой папке проекта), вы сможете найти раздел `authentication`, который выглядит так:

```
<authentication mode="Forms">
  <forms loginUrl="~/Account/Login" timeout="2880"/>
</authentication>
```

Как видите, forms-аутентификация автоматически включена в приложении MVC, созданном на шаблоне `Basic`. Атрибут `loginUrl` сообщает ASP.NET, по какому URL нужно перенаправить пользователей для аутентификации - в данном случае, это страница `~/Account/Login`. Атрибут `timeout` определяет, как долго пользователь остается аутентифицированным после входа в систему. По умолчанию, это 48 часов (2880 минут).

### Примечание

*Основной альтернативой forms-аутентификации является Windows-аутентификация, которая использует учетные данные операционной системы для идентификации пользователей. Это отличное средство, если вы развертываете интранет-приложения и все ваши пользователи находятся в одном домене Windows, но оно не подходит для интернет-приложений.*

Если вы создаете проект приложения MVC, используя опции `Internet Application` или `Mobile Application`, Visual Studio автоматически создаст класс `AccountController`, который будет обрабатывать запросы на аутентификацию с помощью функции членства (membership) ASP.NET.

При создании проекта `SportStore.WebUI` мы выбрали опцию `Basic`, что означает, что у нас включена forms-аутентификация в файле `Web.config`, но нужно создать контроллер, который будет проводить аутентификацию. Это означает, что мы можем реализовать любую модель аутентификации, что очень кстати - функция членства ASP.NET будет излишеством для нашего примера, и для знакомства с функциями безопасности MVC будет достаточно более простой аутентификации.

Для начала мы создадим имя пользователя и пароль, которые будут предоставлять доступ к функциям администрирования SportsStore. В листинге 11-1 показаны изменения, которые мы внесли в раздел аутентификации файла Web.config.

#### Листинг 11-1: Определяем имя пользователя и пароль

```
<authentication mode="Forms">
  <forms loginUrl="~/Account/Login" timeout="2880">
    <credentials passwordFormat="Clear">
      <user name="admin" password="secret" />
    </credentials>
  </forms>
</authentication>
```

Мы не стали усложнять процесс и жестко закодировали имя пользователя (admin) и пароль (secret) в файле Web.config. В этой главе мы хотим сфокусироваться на обеспечении базовой безопасности приложения MVC, так что жесткое кодирование учетных данных нам подойдет.

#### *Внимание!*

*Естественно, мы не рекомендуем развертывать реальное приложение с таким базовым уровнем безопасности. Здесь мы только хотим показать вам основные функции безопасности MVC Framework. Статические и жестко закодированные пароли не пригодны для реальных проектов.*

### Применяем фильтры авторизации

Фильтры – это очень мощная функция MVC Framework. Они представляют собой атрибуты .NET, которые можно применить к методу действия или классу контроллера. Они предоставляют дополнительную логику при обработке запроса. Доступны различные виды фильтров, и вы даже можете создавать пользовательские фильтры, что мы рассмотрим в главе 16. Фильтр, который интересует нас в данный момент – это стандартный фильтр авторизации Authorize. Мы применим его к классу AdminController, как показано в листинге 11-2.

#### Листинг 11-2: Добавляем атрибут Authorize к классу Controller

```
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
namespace SportsStore.WebUI.Controllers
{
  [Authorize]
  public class AdminController : Controller
  {
    private IProductRepository repository;
    public AdminController(IProductRepository repo)
    {
      repository = repo;
    }
    // ...action methods omitted for brevity...
  }
}
```

Атрибут `Authorize`, примененный без параметров, предоставит доступ к методам действия контроллера, если пользователь пройдет аутентификацию. Это означает, что если вы прошли аутентификацию, вы будете автоматически авторизованы для использования функций администрирования. Для SportsStore, где есть только один набор ограниченных методов действия и только один пользователь, будет достаточно такого уровня защиты.

#### Примечание

*Можно применять фильтры к отдельным методам действия или целому контроллеру. При применении фильтра к контроллеру он будет работать так, как если бы вы применили его к каждому методу действия в классе контроллера. В листинге 11-2 мы применяем фильтр `Authorize` к целому классу, так что все методы действий контроллера `Admin` доступны только авторизованным пользователям.*

Чтобы увидеть эффект фильтра `Authorize`, запустите приложение и перейдите по ссылке `/Admin/Index`. Вы увидите ошибку, как показано на рисунке 11-1.

Рисунок 11-1: Эффект фильтра `Authorize`



Если вы попытаетесь получить доступ к методу действия `Index` контроллера `Admin`, MVC Framework обнаружит фильтр `Authorize`. Так как вы не прошли аутентификацию, вы будете перенаправлены на адрес, заданный в разделе forms-аутентификации файла `web.config`: `/Account/Login`. Мы еще не создали контроллер `Account`, и это приводит к ошибке, показанной на рисунке. Однако тот факт, что платформа MVC пыталась создать экземпляр класса `AccountController`, указывает на то, что атрибут `Authorize` работает.

## Создаем провайдер аутентификации

Для использования forms-аутентификации потребуется вызвать два статических метода класса `System.Web.Security.FormsAuthentication`:

- Метод `Authenticate`, который позволяет нам провести валидацию предоставленных пользователем учетных данных.
- Метод `SetAuthCookie`, который добавляет cookie в ответ браузеру, так что пользователю не придется проходить аутентификацию при отправке каждого нового запроса.

Проблема с вызовом статических методов из методов действий заключается в том, что они затрудняют модульное тестирование контроллера. Платформы имитации, такие как Moq, могут создавать имитации только членов экземпляров. Главная причина затруднений заключается в том, что класс `FormsAuthentication` был создан ранее шаблона MVC, поддерживающего модульное тестирование.

Лучший способ решения этой проблемы - отделить контроллер от класса со статическими методами с помощью интерфейса. Дополнительное преимущество этого подхода заключается в том, что он вписывается в более широкий шаблон проектирования MVC и облегчит в дальнейшем переход на другую систему аутентификации.

Мы начнем с определения интерфейса провайдера аутентификации. Создайте новую папку под названием `Abstract` в папке `Infrastructure` проекта `SportsStore.WebUI` и добавьте в нее новый интерфейс под названием `IAuthProvider`. Его содержимое показано в листинге 11-3.

### Листинг 11-3: Интерфейс `IAuthProvider`

```
namespace SportsStore.WebUI.Infrastructure.Abstract {
    public interface IAuthProvider {
        bool Authenticate(string username, string password);
    }
}
```

Теперь мы можем создать реализацию этого интерфейса, которая будет служить оболочкой для статических методов класса `FormsAuthentication`. Создайте еще одну папку в папке `Infrastructure`, назовите ее `Concrete` и определите в ней новый класс под названием `FormsAuthProvider`. Содержание этого класса показано в листинге 11-4.

### Листинг 11-4: Класс `FormsAuthProvider`

```
using System.Web.Security;
using SportsStore.WebUI.Infrastructure.Abstract;
namespace SportsStore.WebUI.Infrastructure.Concrete
{
    public class FormsAuthProvider : IAuthProvider
    {
        public bool Authenticate(string username, string password)
        {
            bool result = FormsAuthentication.Authenticate(username, password);
            if (result)
            {
                FormsAuthentication.SetAuthCookie(username, false);
            }
            return result;
        }
    }
}
```

Реализация метода `Authenticate` вызывает статические методы `FormsAuthentication`, которые мы хотели держать отдельно от контроллера. Последним шагом будет регистрация `FormsAuthProvider` в методе `AddBindings` класса `NinjectControllerFactory`, как показано в листинге 11-5 (изменения выделены жирным шрифтом).

### Листинг 11-5: Добавляем привязку Ninject для провайдера аутентификации

```
using System;
using System.Web.Mvc;
using System.Web.Routing;
using Ninject;
using SportsStore.Domain.Entities;
using SportsStore.Domain.Abstract;
using System.Collections.Generic;
using System.Linq;
using Moq;
using SportsStore.Domain.Concrete;
using System.Configuration;
using SportsStore.WebUI.Infrastructure.Abstract;
using SportsStore.WebUI.Infrastructure.Concrete;
namespace SportsStore.WebUI.Infrastructure
{
    public class NinjectControllerFactory : DefaultControllerFactory
    {
        private IKernel ninjectKernel;
        public NinjectControllerFactory()
        {
            ninjectKernel = new StandardKernel();
            AddBindings();
        }
        protected override IController GetControllerInstance(RequestContext
requestContext, Type controllerType)
        {
            return controllerType == null
                ? null
                : (IController)ninjectKernel.Get(controllerType);
        }
        private void AddBindings()
        {
            ninjectKernel.Bind<IProductRepository>().To<EFProductRepository>();
            EmailSettings emailSettings = new EmailSettings
            {
                WriteAsFile = bool.Parse(ConfigurationManager
                    .AppSettings["Email.WriteAsFile"] ?? "false")
            };
            ninjectKernel.Bind<IOrderProcessor>()
                .To<EmailOrderProcessor>()
                .WithConstructorArgument("settings", emailSettings);
ninjectKernel.Bind<IAuthProvider>().To<FormsAuthProvider>();
        }
    }
}
```

## Создаем контроллер Account

Наша следующая задача – создать контроллер `Account` и метод действия `Login`, упомянутые в файле `Web.config`. На самом деле, мы создадим две версии метода `Login`. Первая будет визуализировать представление, которое содержит подсказку для входа в систему, а другая будет обрабатывать запрос `POST`, после того как пользователь отправит свои учетные данные.

Для начала мы создадим класс модели представления, который будем передавать между контроллером и представлением. Добавьте в папку `Models` проекта `SportsStore.WebUI` новый класс под названием `LoginViewModel` и приведите его содержание в соответствие с листингом 11-6.

#### Листинг 11-6: Класс `LoginViewModel`

```
using System.ComponentModel.DataAnnotations;
namespace SportsStore.WebUI.Models
{
    public class LoginViewModel
    {
        [Required]
        public string UserName { get; set; }
        [Required]
        [DataType(DataType.Password)]
        public string Password { get; set; }
    }
}
```

Этот класс содержит свойства для имени пользователя и пароля, а также использует атрибуты `DataAnnotation`, чтобы указать, что требуются значения обоих свойств. Кроме того, мы используем атрибут `DataType`, чтобы сообщить MVC Framework, как мы хотим отображать редактор свойства `Password`.

Учитывая то, здесь только два свойства, можно было бы попытаться обойтись без модели и использовать `ViewBag` для передачи данных в представление. Тем не менее, лучше определить модель представления, чтобы данные, которые передаются от контроллера к представлению и от механизма связывания в метод действия, были последовательно типизированы. Это облегчит использование шаблонных вспомогательных методов представления.

Далее создайте новый контроллер под названием `AccountController`, как показано в листинге 11-7.

#### Листинг 11-7: Класс `AccountController`

```
using System.Web.Mvc;
using SportsStore.WebUI.Infrastructure.Abstract;
using SportsStore.WebUI.Models;
namespace SportsStore.WebUI.Controllers
{
    public class AccountController : Controller
    {
        IAuthProvider authProvider;
        public AccountController(IAuthProvider auth)
        {
            authProvider = auth;
        }
        public ViewResult Login()
        {
            return View();
        }
        [HttpPost]
        public ActionResult Login(LoginViewModel model, string returnUrl)
        {
            if (ModelState.IsValid)
            {
                if (authProvider.Authenticate(model.UserName, model.Password))
                {
                    return Redirect(returnUrl ?? Url.Action("Index", "Admin"));
                }
                else
                {

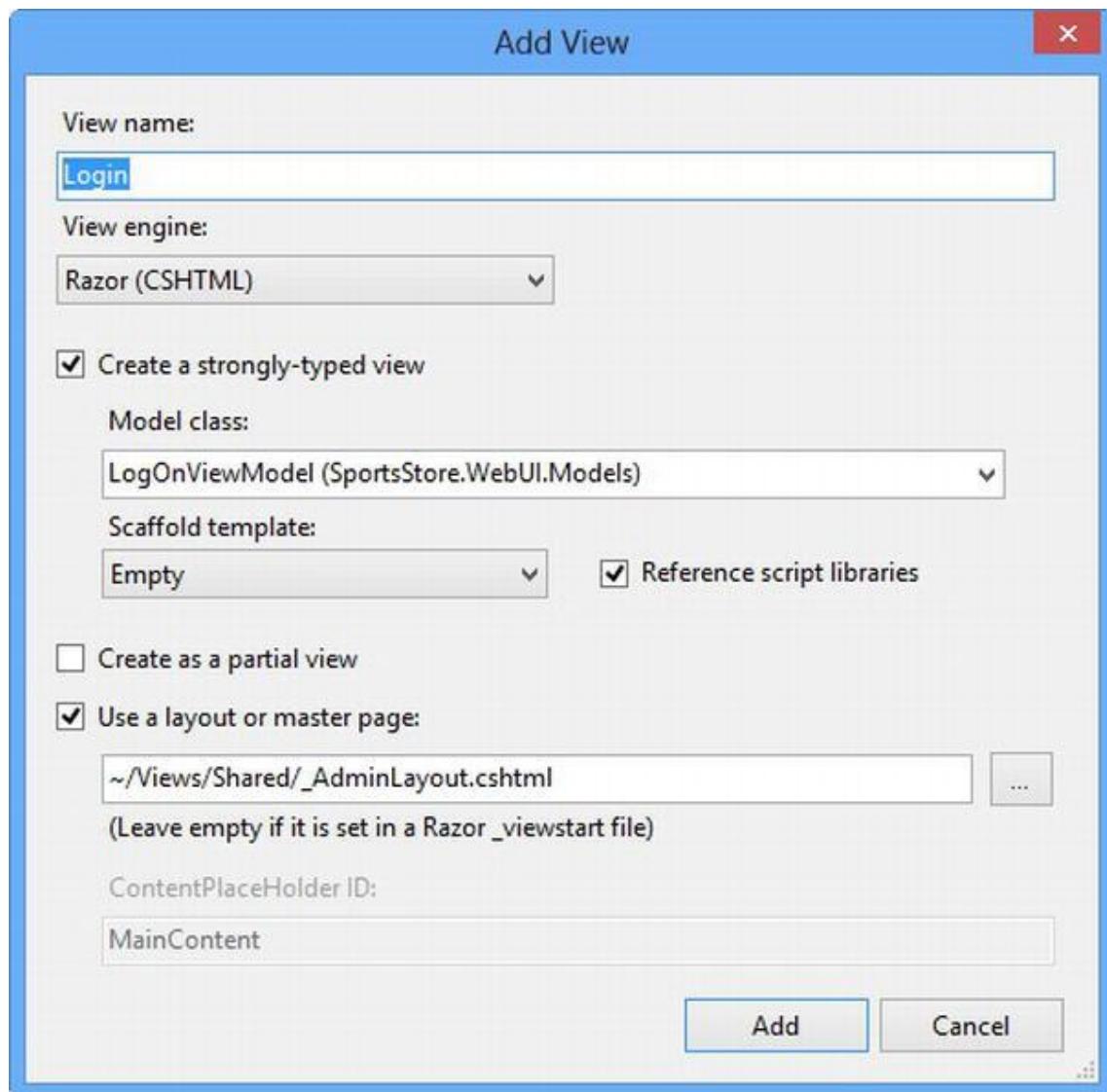
```

```
        ModelState.AddModelError("", "Incorrect username or password");
        return View();
    }
}
else
{
    return View();
}
}
```

## Создаем представление

Кликните правой кнопкой мыши по одному из методов действия в классе контроллера `Account` и выберите `Add View` из контекстного меню. Создайте строго типизированное представление под названием `Login`, которое использует `LoginViewModel` в качестве типа модели представления, как показано на рисунке 11-2. Отметьте флажком опцию `Use a layout` и выберите файл `AdminLayout.cshtml`.

**Рисунок 11-2:** Добавляем представление Login



Нажмите кнопку Add, чтобы создать представление, и отредактируйте разметку в соответствии с листингом 11-8.

#### Листинг 11-8: Представление Login

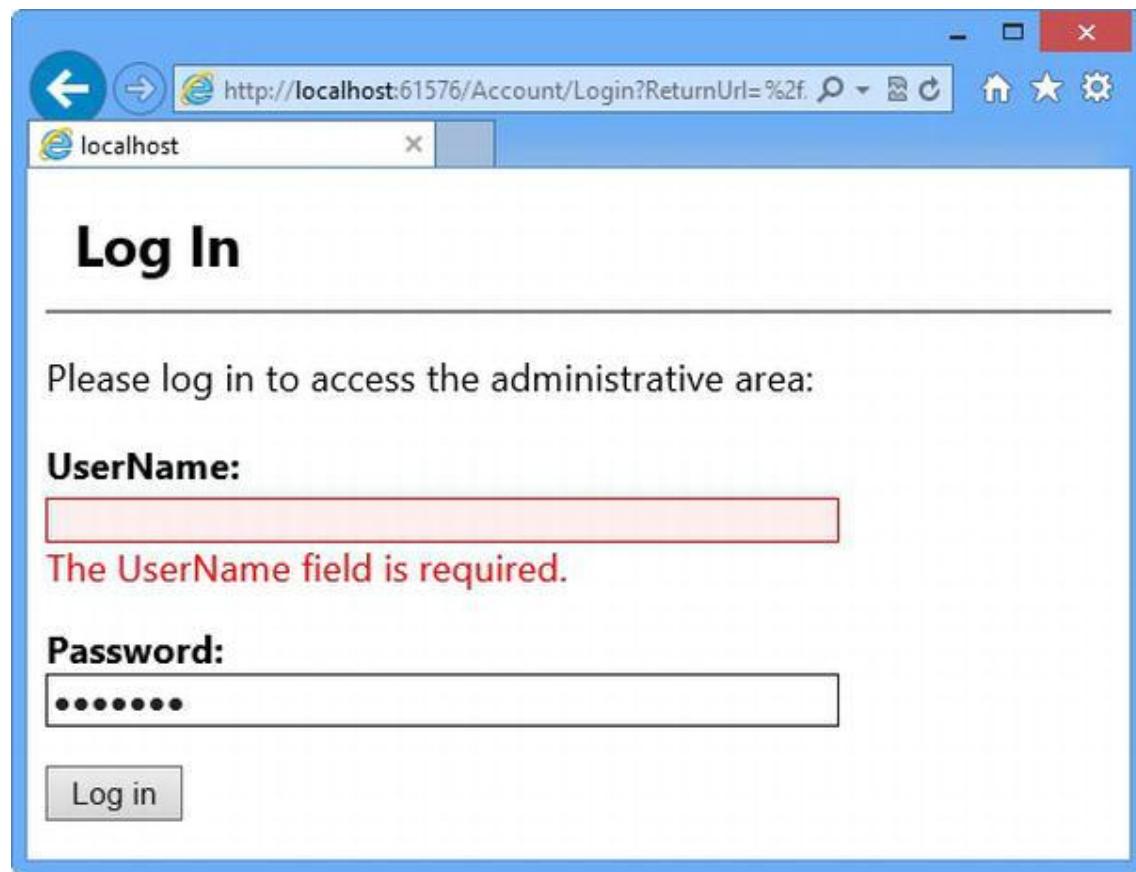
```
@model SportsStore.WebUI.Models.LoginViewModel

{@{
    ViewBag.Title = "Admin: Log In";
    Layout = "~/Views/Shared/_AdminLayout.cshtml";
}
<h1>Log In</h1>
<p>Please log in to access the administrative area:</p>

@using (Html.BeginForm())
{
    @Html.ValidationSummary(true)
    @Html.EditorForModel()
    <p><input type="submit" value="Log in" /></p>
}
```

Чтобы увидеть, как выглядит данное представление, запустите приложение и перейдите по ссылке /Admin/Index, как показано на рисунке 11-3.

Рисунок 11-3: Представление Login



Атрибут `DataType` указывает MVC Framework визуализировать редактор для свойства `Password` как HTML-элемент для ввода пароля, что означает, что символы пароля не видны. Атрибут `Required`, который мы применили к свойствам модели представления, включает валидацию на стороне клиента (мы уже добавили необходимые библиотеки JavaScript в файл `_AdminLayout.cshtml` в главе 10). Пользователи смогут отправить форму только тогда, когда предоставят и имя пользователя, и

пароль. При вызове метода `FormsAuthentication.Authenticate` аутентификация будет проведена на сервере.

#### **Внимание!**

*В принципе, использование валидации данных на стороне клиента - хорошая идея. Это позволяет немного разгрузить сервер и предоставляет пользователю немедленную обратную связь по поводу введенных им данных. Однако не следует проводить аутентификацию на стороне клиента. Как правило, для ее осуществления необходимо будет отправить клиенту действительные учетные данные для проверки введенного пользователем логина и пароля, или, в любом случае, вам придется поверить клиентскому отчету об успешной аутентификации. Аутентификация всегда должна выполняться на сервере.*

---

Когда мы получаем неправильные учетные данные, мы добавляем ошибку в `ModelState` и снова визуализируем представление. Это отображает наше сообщение в области информации о валидации, которую мы создали, вызвав в представлении вспомогательный метод `Html.ValidationSummary`.

#### **Примечание**

*Обратите внимание, что в листинге 11-8 в нашем вызове вспомогательного метода `Html.ValidationSummary` мы выставили параметру `bool` значение `true`. Это исключает отображение сообщений о валидации свойств. Если бы мы этого не сделали, любые сообщения о валидации свойстве дублировались бы в области информации о валидации и рядом с соответствующим элементом ввода.*

## **Модульный тест: аутентификация**

Тестирование контроллера `Account` требует, чтобы мы проверили два вида поведения: пользователь должен быть аутентифицирован, когда предоставлены действительные учетные данные, и не пройти аутентификацию, когда предоставлены неверные учетные данные.

Мы можем выполнить эти тесты, создав имитированную реализацию интерфейса `IAuthProvider` и проверив тип и результат метода контроллера `Login`. Мы создали следующие тесты в новом файле тестов под названием `AdminSecurityTests.cs`:

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;
using SportsStore.WebUI.Controllers;
using SportsStore.WebUI.Infrastructure.Abstract;
using SportsStore.WebUI.Models;
using System.Web.Mvc;
namespace SportsStore.UnitTests
{
    [TestClass]
    public class AdminSecurityTests
    {
        [TestMethod]
        public void Can_Login_With_Valid_Credentials()
        {
            // Arrange - create a mock authentication provider
            Mock<IAuthProvider> mock = new Mock<IAuthProvider>();
            mock.Setup(m => m.Authenticate("admin", "secret")).Returns(true);
            // Arrange - create the view model
```

```

LoginViewModel model = new LoginViewModel
{
    UserName = "admin",
    Password = "secret"
};

// Arrange - create the controller
AccountController target = new AccountController(mock.Object);
// Act - authenticate using valid credentials
ActionResult result = target.Login(model, "/MyURL");
// Assert
Assert.IsInstanceOfType(result, typeof(RedirectResult));
Assert.AreEqual("/MyURL", ((RedirectResult)result).Url);
}

[TestMethod]
public void Cannot_Login_With_Invalid_Credentials()
{
    // Arrange - create a mock authentication provider
    Mock<IAuthProvider> mock = new Mock<IAuthProvider>();
    mock.Setup(m => m.Authenticate("badUser", "badPass")).Returns(false);
    // Arrange - create the view model
    LoginViewModel model = new LoginViewModel
    {
        UserName = "badUser",
        Password = "badPass"
    };
    // Arrange - create the controller
    AccountController target = new AccountController(mock.Object);
    // Act - authenticate using valid credentials
    ActionResult result = target.Login(model, "/MyURL");
    // Assert
    Assert.IsInstanceOfType(result, typeof(ViewResult));
    Assert.IsFalse(((ViewResult)result).ViewData.ModelState.IsValid);
}
}

```

Это обеспечит должную защиту функций администрирования SportsStore. Пользователи смогут получить доступ к этим функциям после того, как предоставят действительные учетные данные и получат файл cookie, который будет прикреплен к последующим запросам.

## Загрузка изображений

Напоследок мы усложним функциональность SportsStore, добавив возможность для администратора загружать изображения товаров и сохранять их в базе данных, чтобы отображать их в каталоге товаров.

## Расширяем базу данных

Откройте окно Visual Studio Database Explorer и перейдите к таблице `Products` в базе, которую мы создали в главе 7. Название соединения для передачи данных может быть изменено на `EFDbContext` – это то название, которое мы назначили соединению в файле `Web.config` в главе 7. Visual Studio может быть немного непоследовательной, когда переименовывает соединения, так что возможно, что вы увидите оригинальное название, которое отображалось при создании соединения.

Щелкните правой кнопкой мыши в таблице и выберите Open Table Definition из контекстного меню. Добавьте определения для двух новых столбцов, которые показаны на рисунке 11-4.

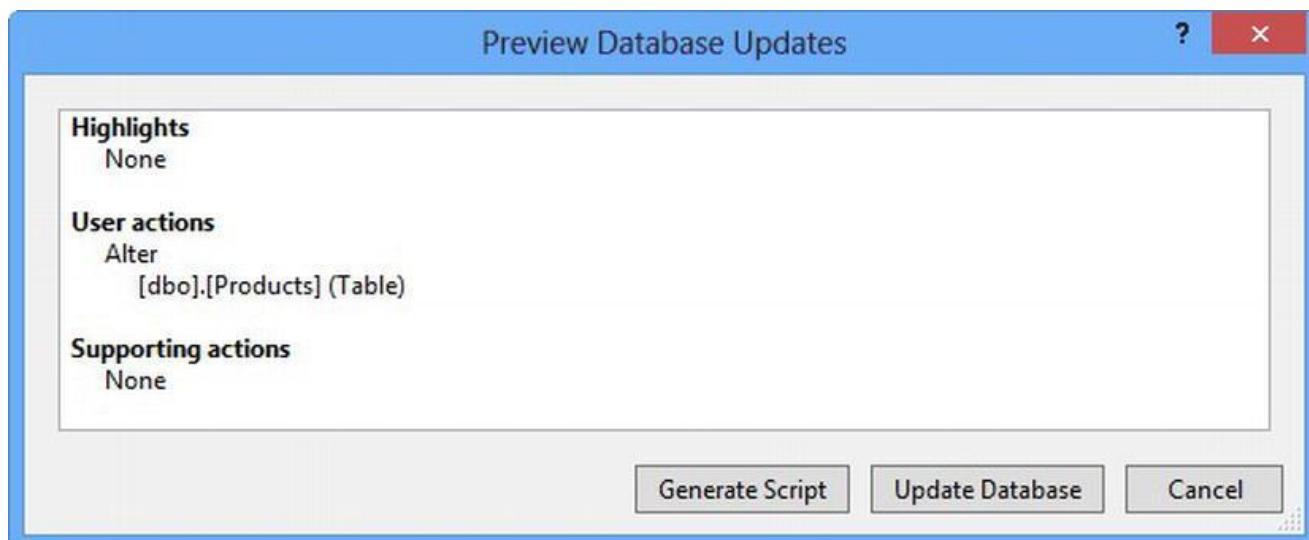
Убедитесь, что вы правильно установили данные столбцов, и не забудьте отметить флашком опцию Allow Nulls для них обоих.

**Рисунок 11-4:** Добавляем новые столбцы в таблицу Products

	Name	Data Type	Allow Nulls	Default
ProductID	int	<input type="checkbox"/>		
Name	nvarchar(100)	<input type="checkbox"/>		
Description	nvarchar(500)	<input type="checkbox"/>		
Category	nvarchar(50)	<input type="checkbox"/>		
Price	decimal(16,2)	<input type="checkbox"/>		
<b>ImageData</b>	<b>varbinary(MAX)</b>	<b><input checked="" type="checkbox"/></b>	<b><input checked="" type="checkbox"/></b>	
<b>ImageMimeType</b>	<b>varchar(50)</b>	<b><input checked="" type="checkbox"/></b>	<b><input type="checkbox"/></b>	

Нажмите кнопку Update, после чего Visual Studio определит, какие операторы SQL нужно отправить в базу данных для ее обновления, и отобразит их в диалоговом окне Preview Database Updates, как показано на рисунке 11-5.

**Рисунок 11-5:** Предварительный просмотр обновлений базы данных



Нажмите кнопку Update Database для создания новых столбцов в базе данных.

## Расширяем доменную модель

Нам нужно добавить два новых поля в класс `Products` проекта `SportsStore.Domain`, которые соответствуют столбцам, добавленным в базу данных. Дополнения выделены жирным шрифтом в листинге 11-9.

### Листинг 11-9: Добавляем свойства в класс `Products`

```
using System.ComponentModel.DataAnnotations;
using System.Web.Mvc;
namespace SportsStore.Domain.Entities
{
    public class Product
    {
        [HiddenInput(DisplayValue = false)]
        public int ProductID { get; set; }

        [Required(ErrorMessage = "Please enter a product name")]
        public string Name { get; set; }

        [Required(ErrorMessage = "Please enter a description")]
        [DataType(DataType.MultilineText)]
        public string Description { get; set; }

        [Required]
        [Range(0.01, double.MaxValue, ErrorMessage = "Please enter a positive price")]
        public decimal Price { get; set; }

        [Required(ErrorMessage = "Please specify a category")]
        public string Category { get; set; }

        public byte[] ImageData { get; set; }

        [HiddenInput(DisplayValue = false)]
        public string ImageMimeType { get; set; }
    }
}
```

Мы не хотим, чтобы какое-либо из этих новых свойств было видимым при визуализации редактора. Для этого мы используем атрибут `HiddenInput` в свойстве `ImageMimeType`. Нам не нужно ничего делать со свойством `ImageData`, потому что платформа не визуализирует редактор для массивов байтов. Она делает это только для "простых" типов, таких как `int`, `string`, `DateTime` и так далее.

#### *Внимание!*

*Убедитесь в том, что имена свойств, которые вы добавляете в класс `Product`, в точности соответствуют названиям новых столбцов в базе данных.*

## Создаем элементы пользовательского интерфейса для загрузки

Далее нам нужно добавить поддержку загрузки файлов. Для этого понадобиться создать пользовательский интерфейс, который будет использоваться администратором для загрузки изображений. Измените представление `Views/Admin/Edit.cshtml` в соответствии с листингом 11-10 (дополнения выделены жирным шрифтом).

### Листинг 11-10: Добавляем поддержку изображений

```
@model SportsStore.Domain.Entities.Product
```

```

@{
    ViewBag.Title = "Admin: Edit " + @Model.Name;
    Layout = "~/Views/Shared/_AdminLayout.cshtml";
}

<h1>Edit @Model.Name</h1>
@using (Html.BeginForm("Edit", "Admin",
    FormMethod.Post, new { enctype = "multipart/form-data" })) {
    @Html.EditorForModel()
    <div class="editor-label">Image</div>
    <div class="editor-field">
        @if (Model.ImageData == null) {
            @:None
        }
        else {
            
        }
        <div>Upload new image:
            <input type="file" name="Image" /></div>
    </div>
    <input type="submit" value="Save" />
    @Html.ActionLink("Cancel and return to List", "Index")
}

```

Вы, возможно, уже знаете, что браузеры будут загружать файлы должным образом только тогда, когда атрибут `enctype` в HTML-элементе `form` содержит значение `multipart/form-data`. Другими словами, для успешной загрузки элемент `form` должен выглядеть следующим образом:

```

<form action="/Admin/Edit" enctype="multipart/form-data" method="post">
    ...
</form>

```

Без атрибута `enctype` браузер будет передавать только имя файла, а не его содержимое, что для нас совершенно бесполезно. Чтобы гарантировать наличие атрибута `enctype`, мы должны использовать перегруженный вспомогательный метод `Html.BeginForm`, который позволяет указать HTML-атрибуты, например:

```

@using (Html.BeginForm("Edit", "Admin", FormMethod.Post,
    new { enctype = "multipart/form-data" })) {

```

Обратите внимание, что если свойство `ImageData` отображаемого объекта `Product` не содержит `null`, мы добавляем элемент `img` и устанавливаем в качестве его источника результат вызова метода действия `GetImage` контроллера `Product`. Скоро мы это реализуем.

## Сохраняем изображения в базе данных

Нам нужно расширить версию POST метода действия `Edit` класса `AdminController` так, чтобы мы могли принимать загруженное изображение и сохранять его в базе данных. Необходимые изменения показаны в листинге 11-11.

### Листинг 11-11: Обработка изображения в классе `AdminController`

```

[HttpPost]
public ActionResult Edit(Product product, HttpPostedFileBase image)
{
    if (ModelState.IsValid)
    {
        if (image != null)

```

```

    {
        product.ImageMimeType = image.ContentType;
        product.ImageData = new byte[image.ContentLength];
        image.InputStream.Read(product.ImageData, 0, image.ContentLength);
    }
    repository.SaveProduct(product);
    TempData["message"] = string.Format("{0} has been saved", product.Name);
    return RedirectToAction("Index");
}
else
{
    // there is something wrong with the data values
    return View(product);
}
}

```

Мы добавили в метод `Edit` новый параметр, с помощью которого MVC Framework передает нам данные загруженного файла. Мы проверяем значение этого параметра, и если он не содержит `null`, то копируем данные и тип MIME из параметра в объект `Product`, чтобы сохранить их в базе данных.

#### **Примечание**

*Понадобится обновить модульные тесты, чтобы отразить новый параметр в листинге 11-11. Передача параметра со значением null позволит тестам скомпилироваться.*

---

Мы также должны обновить класс `EFProductRepository` в проекте `SportsStore.Domain` и гарантировать, что значения, присвоенные свойствам `ImageData` и `ImageMimeType`, сохраняются в базе данных. В листинге 11-12 показаны необходимые изменения в методе `SaveProduct`.

**Листинг 11-12:** Гарантируем, что данные изображений сохраняются в базе данных

```

public void SaveProduct(Product product)
{
    if (product.ProductID == 0)
    {
        context.Products.Add(product);
    }
    else
    {
        Product dbEntry = context.Products.Find(product.ProductID);
        if (dbEntry != null)
        {
            dbEntry.Name = product.Name;
            dbEntry.Description = product.Description;
            dbEntry.Price = product.Price;
            dbEntry.Category = product.Category;
            dbEntry.ImageData = product.ImageData;
            dbEntry.ImageMimeType = product.ImageMimeType;
        }
    }
    context.SaveChanges();
}

```

#### **Реализуем метод действия GetImage**

В листинге 11-10 мы добавили элемент `img`, содержание которого было получено с помощью метода действия `GetImage`. Теперь мы его реализуем, чтобы можно было отображать изображения,

содержащиеся в базе данных. В листинге 11-13 показан метод, который мы добавили в класс `ProductController`.

#### Листинг 11-13: Метод действия `GetImage`

```
public FileContentResult GetImage(int productId)
{
    Product prod = repository.Products.FirstOrDefault(p => p.ProductID == productId);
    if (prod != null)
    {
        return File(prod.ImageData, prod.ImageMimeType);
    }
    else
    {
        return null;
    }
}
```

Этот метод пытается найти товар, который соответствует указанному в параметре ID. Он возвращает класс `FileContentResult`, когда мы хотим вернуть файл браузеру клиента, и экземпляры создаются с помощью метода `File` базового класса контроллера. Мы обсудим различные типы результатов, которые можно возвращать из методов действий, в главе 15.

#### Модульный тест: получение изображений

Мы хотим убедиться, что метод `GetImage` возвращает из хранилища правильный тип MIME, и гарантировать, что данные не возвращаются, если мы запрашиваем несуществующий ID товара. Вот тестовые методы, которые мы добавили в новый файл тестов под названием `ImageTests.cs`:

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using SportsStore.WebUI.Controllers;
using System.Linq;
using System.Web.Mvc;
namespace SportsStore.UnitTests
{
    [TestClass]
    public class ImageTests
    {
        [TestMethod]
        public void Can_Retrieve_Image_Data()
        {
            // Arrange - create a Product with image data
            Product prod = new Product
            {
                ProductID = 2,
                Name = "Test",
                ImageData = new byte[] { },
                ImageMimeType = "image/png"
            };
            // Arrange - create the mock repository
            Mock<IProductRepository> mock = new Mock<IProductRepository>();
            mock.Setup(m => m.Products).Returns(new Product[] {
                new Product { ProductID = 1, Name = "P1" },
                prod,
                new Product { ProductID = 3, Name = "P3" }
            }.AsQueryable());
            // Arrange - create the controller
            ProductController target = new ProductController(mock.Object);
            // Act - call the GetImage action method
        }
    }
}
```

```

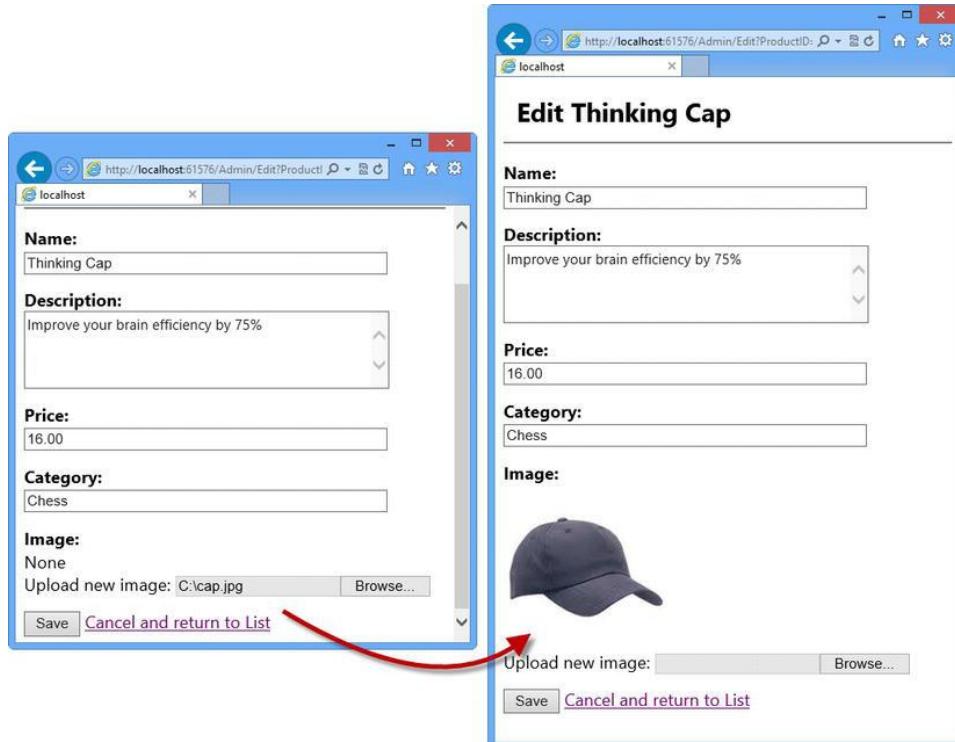
        ActionResult result = target.GetImage(2);
        // Assert
        Assert.IsNotNull(result);
        Assert.IsInstanceOfType(result, typeof(FileResult));
        Assert.AreEqual(prod.ImageMimeType, ((FileResult)result).ContentType);
    }
    [TestMethod]
    public void Cannot_Retrieve_Image_Data_For_Invalid_ID()
    {
        // Arrange - create the mock repository
        Mock<IProductRepository> mock = new Mock<IProductRepository>();
        mock.Setup(m => m.Products).Returns(new Product[] {
            new Product { ProductID = 1, Name = "P1" },
            new Product { ProductID = 2, Name = "P2" }
        }.AsQueryable());
        // Arrange - create the controller
        ProductController target = new ProductController(mock.Object);
        // Act - call the GetImage action method
        ActionResult result = target.GetImage(100);
        // Assert
        Assert.IsNull(result);
    }
}
}

```

Когда у нас есть действительный ID товара, тест гарантирует то, что мы получаем результат `FileResult` из метода действия и тип содержимого соответствует типу имитированных данных. Класс `FileResult` не дает нам доступ к двоичному содержимому файла, так что придется довольствоваться не вполне совершенным тестом. Когда мы запрашиваем недействительный ID товара, мы просто проверяем, что результат будет содержать `null`.

Теперь администратор может загружать изображения для товаров. Вы можете попробовать сделать это сами, запустив приложение, перейдя по ссылке в `/Admin/Index` и отредактировав один из товаров. Пример показан на рисунке 11-6.

**Рисунок 11-6:** Добавляем изображение к списку товаров



## Выводим изображения товаров

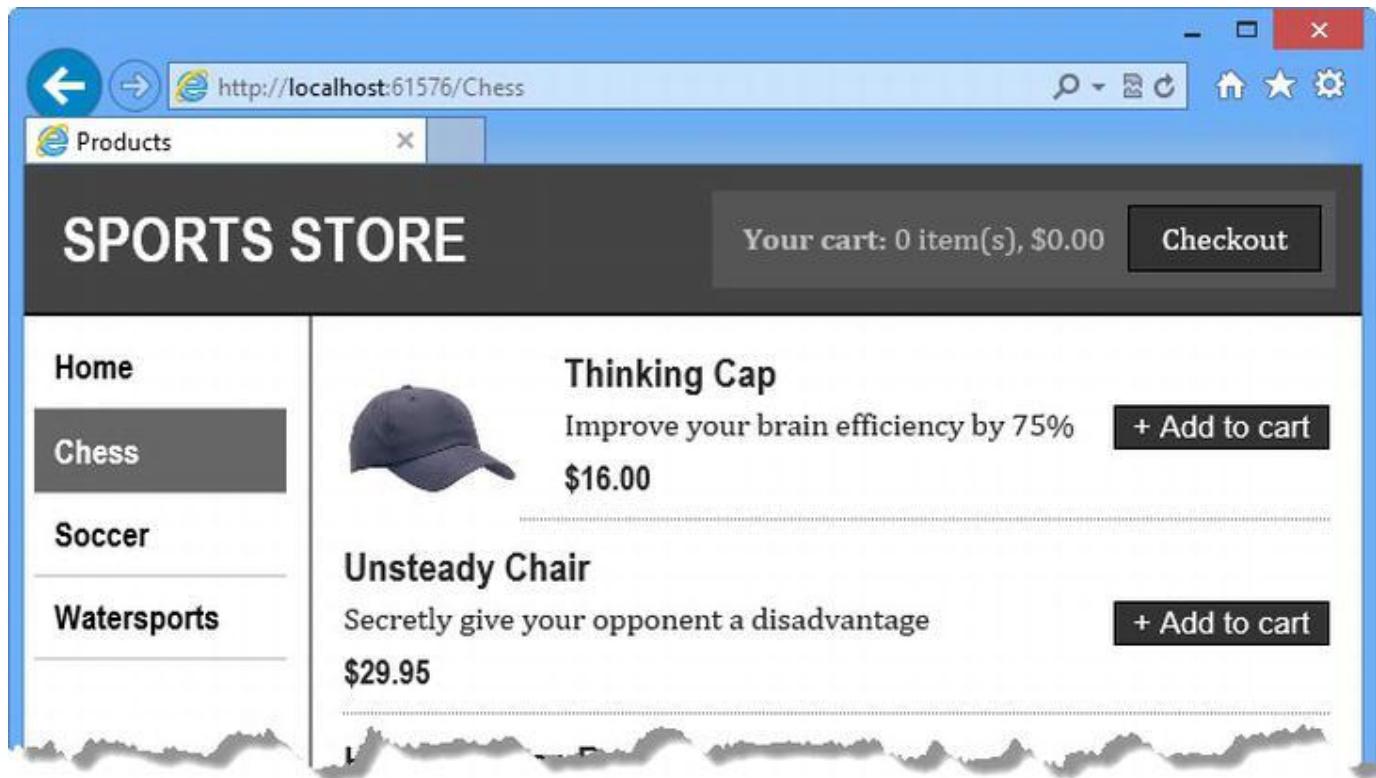
Теперь нам остается только визуализировать изображения рядом с описаниями товаров в каталоге. Отредактируйте представление Views/Shared/ProductSummary.cshtml и внесите в него изменения, выделенные жирным шрифтом в листинге 11-14.

**Листинг 11-14:** Отображаем изображения в каталоге товаров

```
@model SportsStore.Domain.Entities.Product
<div class="item">
    @if (Model.ImageData != null) {
        <div style="float: left; margin-right: 20px">
            
        </div>
    }
    <h3>@Model.Name</h3>
    @Model.Description
    <div class="item">
        @using (Html.BeginForm("AddToCart", "Cart")) {
            @Html.HiddenFor(x => x.ProductID)
            @Html.Hidden("returnUrl", Request.Url.PathAndQuery)
            <input type="submit" value="+ Add to cart" />
        }
    </div>
    <h4>@Model.Price.ToString("c")</h4>
</div>
```

Когда эти изменения применены, пользователи при просмотре каталога будут видеть изображения как часть описания товара, как показано на рисунке 11-7.

**Рисунок 11-7:** Выводим изображения товаров



## Резюме

В этой и предыдущих главах мы продемонстрировали, как с помощью ASP.NET MVC Framework можно создать реалистичное приложение для электронной коммерции. В этом расширенном примере мы рассмотрели многие ключевые возможности платформы: контроллеры, методы действий, маршрутизацию, представления, связывание данных, метаданные, валидацию, макеты, аутентификацию и многое другое. Вы также видели, как можно использовать такие важные технологии, связанные с MVC, как Entity Framework, Ninject, MOQ и поддержка модульного тестирования в Visual Studio.

В итоге мы получили приложение с чистой, компонентно-ориентированной архитектурой, в которой разделены обязанности, а кодовую базу будет легко расширять и поддерживать. Во второй части этой книги мы подробнее рассмотрим каждый компонент MVC Framework, чтобы дать вам детальное описание возможностей платформы.

# ASP.NET MVC 4 в деталях

И вот вы узнали о том, почему существует ASP.NET MVC, и получили представление о его архитектуре и методах. Все это мы изучили на примере реального коммерческого приложения. Теперь пришло время открыть капот и взглянуть на все детали этой машины.

В части 2 этой книги мы подробно рассмотрим механизм фреймворка. Мы начнем с изучения структуры ASP.NET MVC приложения и способа, которым обрабатываются запросы. Затем сосредоточимся на отдельных функциях, таких как маршрутизация (главы 13 и 14), контроллеры и методы действий (главы 15-17), MVC представления и система вспомогательных методов (главы 18-21), и на том, как MVC работает с доменной моделью (главы 22 и 23).

В последних главах этой книги мы покажем вам возможности MVC, которые упрощают разработку на стороне клиента (главы 24 и 25), а также мы расскажем, как подготовить и развернуть MVC приложение (глава 26).

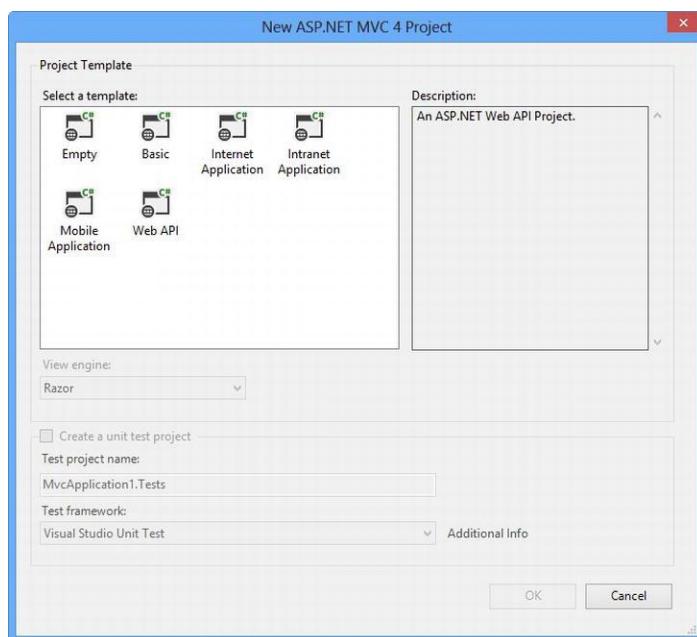
## Обзор MVC проектов

Мы собираемся предоставить некоторую дополнительную информацию, прежде чем начать погружаться в детали конкретных функций MVC фреймворка. В этой главе дается обзор структуры и характера ASP.NET MVC приложения, включая структуру проекта по умолчанию и соглашение об именах, которому необходимо следовать.

## Работа с MVC проектами Visual Studio

Когда вы создаете новый MVC проект, Visual Studio дает вам возможность выбора между различными отправными точками: **Empty**, **Basic**, **Internet Application**, **Intranet Application**, **Mobile Application** и **Web API**, как показано на рисунке 12-1.

Рисунок 12-1: Выбор начальной конфигурации MVC проекта



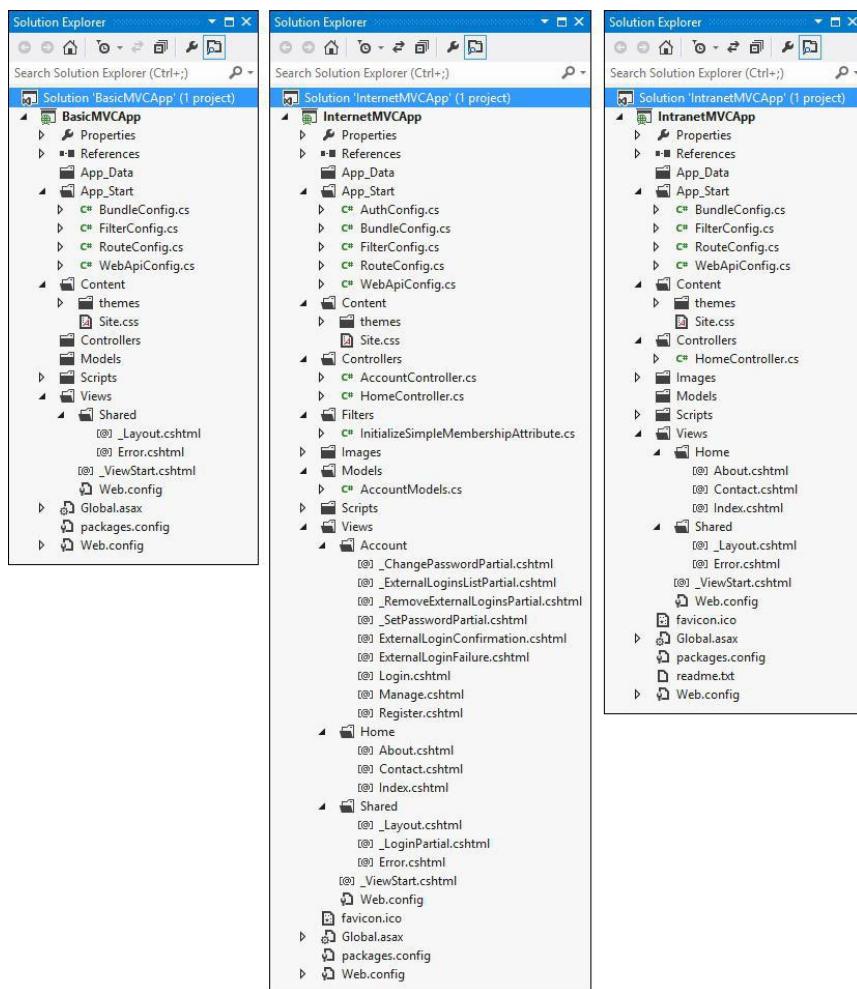
Мы уже показали вам первые два шаблона в предыдущих главах. Шаблон проекта **Empty** мы использовали в главе 2 для приложения RSVP. Он содержит только необходимый минимум файлов, необходимых для MVC фреймворка.

Мы использовали шаблон проекта **Basic** в главе 7, когда мы создали приложение SportsStore. Его структура дополнена, по сравнению с шаблоном **Empty**, некоторыми макетами, файлами JavaScript и некоторыми CSS стилями, используемыми для элементов HTML форм и валидации. Шаблон **Basic** мы используем в наших собственных проектах, потому что скриптовые файлы и другие дополнения полезны, но мы все же можем сами реализовывать важные части проекта, как мы делали для приложения SportsStore.

Шаблоны **Internet Application** и **Intranet Application** заполнены более полно, тут используются различные механизмы аутентификации, которые подходят для интернет и интранет приложений. Шаблон **Mobile Application** является вариацией шаблона **Internet Application**, и он оптимизирован для мобильных устройств (мы объясним новые функции MVC 4 для мобильных устройств в главе 24). И, наконец, шаблон **Web API** создает проект, который поможет вам начать работу с новой MVC 4 Web API функцией, которую мы объясним в главе 25.

Вы можете увидеть разницу между тремя из этих шаблонов на рисунке 12-2, где мы показали содержимое Solution Explorer для шаблонов **Basic**, **Internet Application** и **Intranet Application**.

**Рисунок 12-2:** Начальное содержание, созданное шаблонами **Empty**, **Internet Application** и **Intranet Application**



Вы видите, что проект **Internet Application** является самым сложным, это потому что он реализует всю систему пользовательской аутентификации, которая требует много различных представлений. Проект **Intranet Application** может работать с Windows аутентификацией для обработки задач, связанных, например, со сменой пароля. А шаблон **Basic** по умолчанию вообще не обеспечивает аутентификацию.

В основном мы используем шаблоны `Empty` и `Basic` и рекомендуем вам делать то же самое. По умолчанию функционал в других шаблонах является слишком общим, чтобы быть полезным, и требует больших трудовых затрат. Мы получаем лучшие результаты, когда строим наши проекты с нуля.

Какой бы шаблон вы ни выбрали, вы заметите, что в результате у проектов очень похожие структуры. Некоторые из элементов MVC проекта имеют особые роли, они жестко вписаны в ASP.NET или MVC фреймворк. Другие затрагивают соглашение по именам. Мы описали каждый из этих файлов и папок в таблице 12-1.

**Таблица 12-1:** Элементы MVC проекта

Папка или файл	Описание	Примечание
/App_Data	В эту папку вы кладете закрытые данные, такие как XML-файлы или базы данных, если вы используете SQL Server Express, SQLite или другие файловые хранилища.	IIS не будет обрабатывать содержимое этой папки.
/App_Start	В этой папке содержатся некоторые основные параметры конфигурации для вашего проекта, в том числе определение роутов, фильтры, связки.	Мы описываем роуты в главе 13, фильтры в главе 16 и связки в главе 24.
/bin	Здесь размещается скомпилированная сборка для вашего MVC приложения, наряду с любыми связанными сборками, которые не в GAC.	IIS не будет обрабатывать содержимое этой директории. Вы не увидите bin директорию в окне Solution Explorer, пока не нажмете кнопку Show All Files. Поскольку это бинарные файлы, созданные при компиляции, вам обычно не следует хранить их в системе управления.
/Content	Здесь вы храните статический контент, как CSS стили и рисунки.	Это соглашение, но оно не обязательно. Вы можете хранить статический контент где-то еще.
/Controllers	Здесь хранятся классы контроллеров.	Это соглашение. Вы можете разместить классы контроллера где угодно, потому что все они скомпилированы в одну сборку.
/Models	Здесь вы храните классы моделей представления и доменной модели, хотя все приложения, кроме простейших, выигрывают от определения доменной модели в специальном проекте, как мы продемонстрировали в SportsStore.	Это соглашение. Вы можете определить классы модели в любом месте проекта или в отдельном проекте.
/Scripts	В этой директории содержатся JavaScript библиотеки для вашего проекта. Visual	Это соглашение. Вы можете поместить скриптовые файлы в

Папка или файл	Описание	Примечание
	Studio добавляет jQuery и некоторое другие популярные библиотеки.	любом месте, так как они в действительности являются просто еще одним типом статического контента. См. главу 24 для получения дополнительной информации об управлении скриптовыми файлами.
/Views	В этой директории содержатся представления и частичные представления, как правило, сгруппированные в папки с именем контроллера, с которым они связаны.	Файл /Views/Web.config не дает IIS обрабатывать содержимое этих директорий. Представления должны демонстрироваться через метод действия.
/Views/Shared	В этой директории содержатся макеты и представления, которые не являются конкретными для отдельного контроллера.	
/Views/Web.config	Это <i>не</i> конфигурационный файл для вашего приложения. Он содержит настройки, необходимые для того, чтобы представления работали ASP.NET, и предотвращает то, чтобы представления не обрабатывались IIS и пространствами имен, импортированными в представления по умолчанию.	
/Global.asax	Это глобальный класс ASP.NET приложения. Его класс с выделенным кодом (Global.asax.cs) является местом для регистрации настройки маршрутизации, а также настройки любого кода для запуска при инициализации приложения или завершении работы, а также при получении необработанных исключений.	Файл Global.asax играет такую же роль в MVC приложениях, как и в приложениях Web Forms.
/Web.config	Это конфигурационный файл для вашего приложения. Мы лучше поясним его роль далее в этой главе.	Файл Web.config играет такую же роль в MVC приложениях, как и в приложениях Web Forms.

В таблице 12-2 представлены файлы и папки, которые имеют особенное значение, если они присутствуют в MVC проекте.

**Таблица 12-2:** Дополнительные элементы MVC проекта

Папка или файл	Описание
/Areas	Области (areas) являются одним из способов разбития больших приложений на более мелкие куски. Мы расскажем об областях в главе 13.
/App_GlobalResources, /App_LocalResources	В них хранятся исходные файлы, используемые для локализации страниц Web Forms.
/App_Browsers	Эта папка содержит XML файлы .browser, которые описывают, как

Папка или файл	Описание
	идентифицировать определенные веб-браузеры, и на что способны эти браузеры (например, поддерживают ли они JavaScript).
/App_Themes	Эта папка содержит темы Web Forms (в том числе файлы .skin), которые влияют на то, как отображаются элементы управления Web Forms.

#### *Примечание*

*За исключением /Areas, элементы в таблице 12-2 являются частью ядра платформы ASP.NET и не особенно актуальны для MVC приложений. Адам рассказывает об основных функциях ASP.NET в своих книгах Applied ASP.NET 4.5 in Context и Pro ASP.NET 4.5, опубликованных Apress.*

---

## Понимание MVC соглашений

Есть два вида соглашений для MVC проектов. Первое из них – это действительно только предложение о том, как вы можете структурировать проект. Например, по соглашению JavaScript файлы размещаются в папке Scripts. Это место, где другие MVC разработчики в первую очередь будут их искать, а также именно сюда Visual Studio помещает исходные файлы JavaScript для нового MVC проекта. Но вы можете переименовать папку Scripts или удалить ее полностью и сохранять ваши скрипты где угодно. Это не помешает MVC фреймворку запускать приложение.

Другой вид соглашения вытекает из принципа *соглашения о конфигурации*, которое было одним из основных пунктов продажи, сделавшим Ruby on Rails таким популярным. Соглашение о конфигурации означает, что вам не нужно, например, явно настраивать связь между контроллерами и представлениями. Вы просто следите определенным именованиям для ваших файлов, и все работает. Тут существует меньше гибкости в изменении структуры проекта, если вы имеете дело с таким соглашением. О соглашениях более подробно рассказывается в следующих разделах.

#### *Совет*

*Все соглашения могут быть изменены, если вы используете пользовательский движок представления, который мы рассмотрим в главе 18. Но не стоит этим слишком увлекаться: по большей части вы будете использовать эти соглашения в своих MVC проектах.*

---

## Следуя соглашениям для классов контроллеров

Классы контроллеров должны иметь имена, которые заканчиваются на Controller, например, ProductController, AdminController и HomeController.

При обращении к контроллеру из другой части проекта, например, при использовании вспомогательного метода, необходимо указать первую часть имени (как Product), а MVC фреймворк автоматически добавит к имени Controller и начинает искать класс контроллера.

### *Совет*

*Вы можете изменить это поведение, если создадите свою собственную реализацию интерфейса `IControllerFactory`, который мы опишем в главе 17.*

---

## **Следуя соглашениям для представления**

Представления и частичные представления находятся в папке `/Views/Controllername`. Например, представление, связанное с `ProductController`, будет находиться в папке `/Views/Product`.

### *Совет*

*Обратите внимание, что мы опускаем часть класса `Controller` из папки `Views`: Мы используем папку `/Views/Product`, а не `/Views/ProductController`. Это может показаться нелогичным на первый взгляд, но очень скоро вы к этому привыкнете.*

---

MVC фреймворк ожидает, что представление по умолчанию для метода действия должно быть названо после этого метода. Например, представление по умолчанию, связанное с методом действия `List` следует называть `List.cshtml`. Таким образом ожидается, что для метода действия `List` в классе `ProductController` представление по умолчанию будет называться `/Views/Product/List.cshtml`.

Представление по умолчанию используется, если вы возвращаете в методе действия результат вызова метода `View`:

```
return View();
```

Вы можете указать другое представление по имени:

```
return View("MyOtherView");
```

Обратите внимание, что мы не включаем расширение имени файла или путь к представлению. При поиске представления MVC фреймворк смотрит в папку с именем контроллера, а затем в папку `/Views/Shared`. Это означает, что мы можем хранить представления, которые будут использоваться более чем одним контроллером, в папке `/Views/Shared`, и фреймворк их найдет.

## **Следуя соглашениям для макетов**

Соглашение по именованию для макетов заключается в том, что перед именем файла стоит знак подчеркивания (`_`), а файлы макетов помещаются в папку `/Views/Shared`. Visual Studio создает макет `_Layout.cshtml`, если используется любой шаблон, кроме `Empty`. Этот макет применяется ко всем представлениям по умолчанию через файл `/Views/_ViewStart.cshtml`.

Если вы не хотите, чтобы макет по умолчанию применялся ко всем представлениям, вы можете изменить настройки в `_ViewStart.cshtml` (или полностью удалить файл), чтобы указать другой макет для представления:

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

Или вы можете отключить любой макет для данного представления:

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

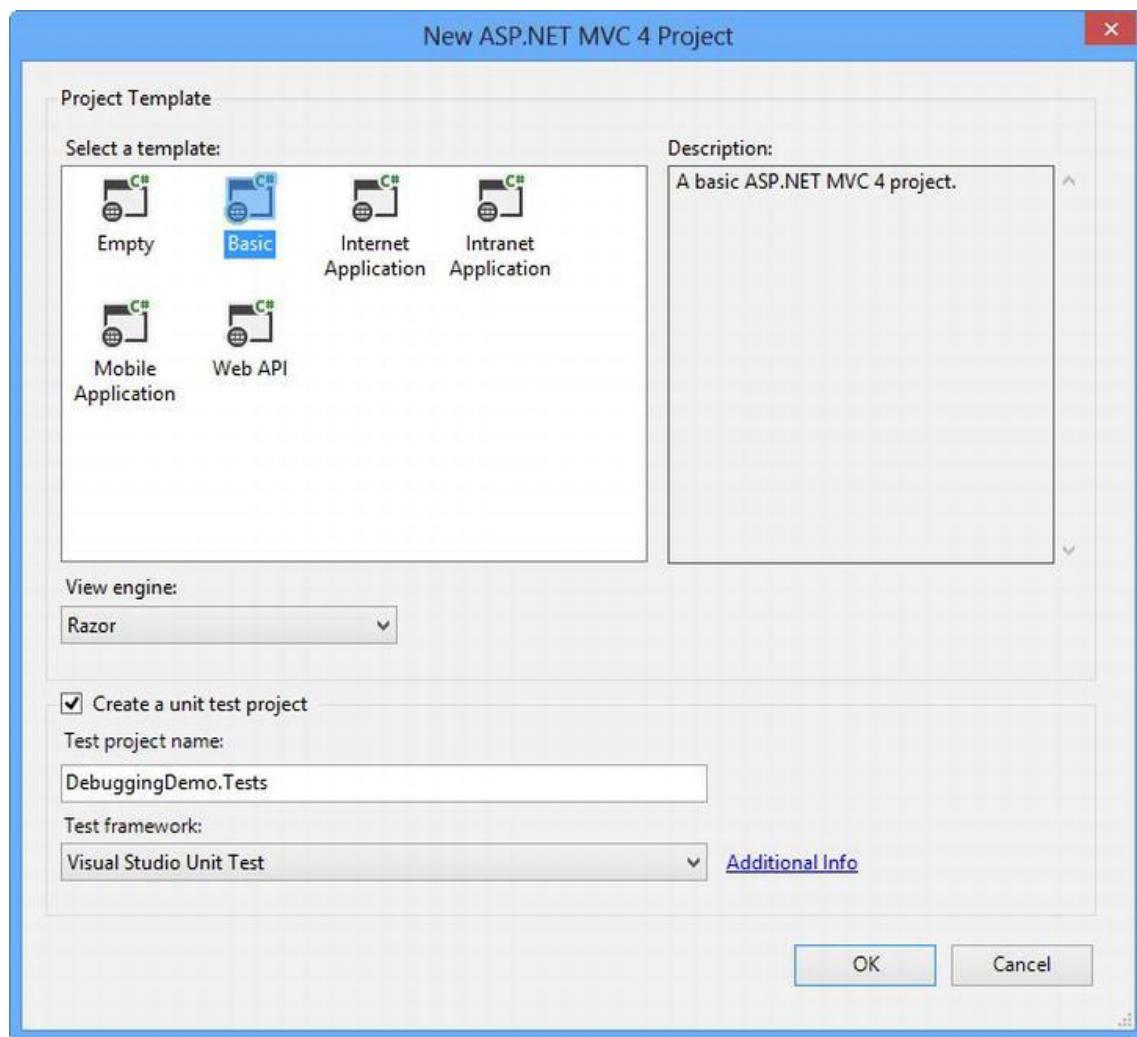
## Отладка MVC приложений

Вы можете провести отладку ASP.NET MVC приложения точно так же, как и отладку ASP.NET Web Forms приложения. Отладчик (дебаггер) Visual Studio представляет собой мощный и гибкий инструмент, с большим количеством функций и задач. Мы можем лишь поверхностно охватить данную тему в этой книге, но и в последующих разделах мы покажем вам, как настроить отладчик и выполнять различные виды отладки для вашего MVC проекта.

### Создание проекта

Чтобы продемонстрировать использование отладчика, мы создали новый MVC проект при помощи шаблона Basic. Мы назвали наш проект DebuggingDemo и проверили возможность создавать проект модульного теста, как показано на рисунке 12-3.

**Рисунок 12-3:** Создание проекта DebuggingDemo



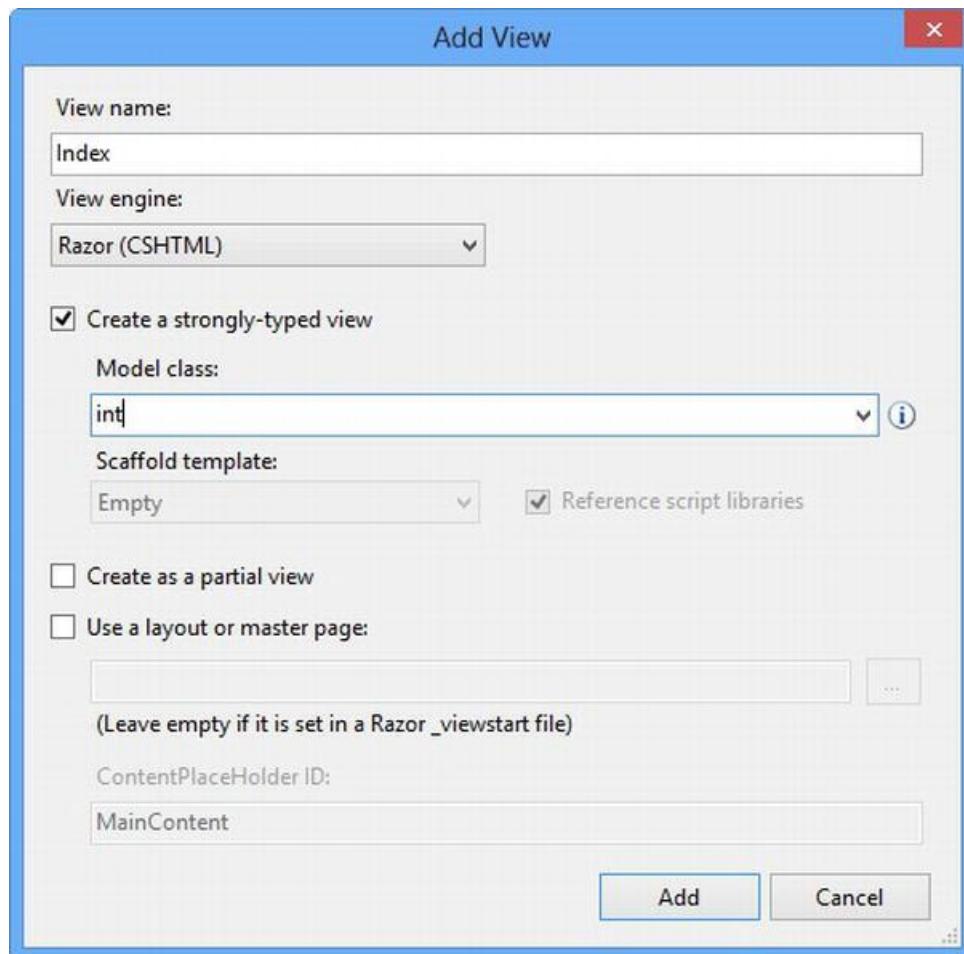
Создайте новый контроллер `Home` и задайте ему содержание в соответствии с листингом 12-1. Выражения в методах действия не делают ничего интересного, но мы будем использовать их, чтобы продемонстрировать различные функции отладки.

**Листинг 12-1:** Начальное содержание контроллера `Home`

```
using System.Web.Mvc;
namespace DebuggingDemo.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            int firstVal = 10;
            int secondVal = 5;
            int result = firstVal / secondVal;
            ViewBag.Message = "Welcome to ASP.NET MVC!";
            return View(result);
        }
    }
}
```

Кликните правой кнопкой мыши по методу действия `Index`, выберите `Add View` из всплывающего меню и создайте новое представление. Установите класс `Model` на `int` и снимите галочку с опции `Use a layout or master page`, как показано на рисунке 12-4.

**Рисунок 12-4:** Создание представления



Нажмите кнопку Add и отредактируйте содержание представления, чтобы оно соответствовало листингу 12-2. Это очень простое представление, которое просто отображает значение модели представления и ViewBag свойство, которое мы определили в методе действия Index.

#### Листинг 12-2: Начальное содержание представления Index.cshtml

```
@model int
 @{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link href("~/Content/Site.css" rel="stylesheet" type="text/css" />
    <title>Index</title>
</head>
<body>
    <h2 class="message">@ViewData["Message"]</h2>
    <p>
        The calculation result value is: @Model
    </p>
</body>
</html>
```

Последний подготовительный шаг, который нам нужно сделать, заключается в том, чтобы добавить стиль в файл /Content/Site.css, как показано в листинге 12-3.

#### Листинг 12-3: Добавление стиля в файл /Content/Site.css

```
...
.message {
    font-size: 20pt;
    text-decoration: underline;
}
...
```

### Запуск отладчика Visual Studio

Отладка MVC приложений управляется по-другому, нежели отладка других приложений Visual Studio. Visual Studio настраивает отладку по умолчанию, но вам полезно знать, как изменить конфигурацию.

Важная настройка находится в файле Web.config в корневой папке проекта и может быть найдена в элементе system.web, как показано в листинге 12-4.

#### Листинг 12-4: Атрибут debug в файле Web.config

```
...
<system.web>
    <httpRuntime targetFramework="4.5" />
    <compilation debug="true" targetFramework="4.5" />
    <pages>
        <namespaces>
            <add namespace="System.Web.Helpers" />
            <add namespace="System.Web.Mvc" />
            <add namespace="System.Web.Mvc.Ajax" />
            <add namespace="System.Web.Mvc.Html" />
            <add namespace="System.Web.Routing" />
```

```

<add namespace="System.Web.WebPages" />
</namespaces>
</pages>
</system.web>
...

```

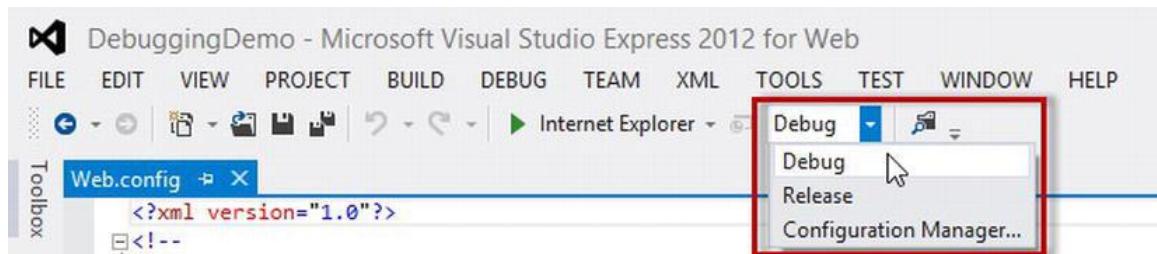
Часть компиляции MVC проекта приходится на тот момент, когда приложение запускается в IIS, и поэтому вы должны убедиться, что атрибут debug установлен на true во время процесса разработки. Это гарантирует, что отладчик сможет работать с файлами классов, которые создаются по требованию компиляции.

#### Внимание

Не разворачивайте приложение на производственном сервере без отключения параметры debug. Мы объясним, почему этого нельзя делать, в главе 26.

Кроме работы с файлом Web.config мы должны убедиться, что Visual Studio включает информацию об отладке в файлы классов, которые она создает. Это не критично, но могут возникнуть проблемы, если различные настройки отладки не синхронизированы. Убедитесь, в панели Visual Studio выбрана конфигурация Debug, как показано на рисунке 12-5.

**Рисунок 12-5:** Выбор конфигурации Debug



Для отладки MVC приложения выберите команду Start Debugging из Visual Studio меню Debug или нажмите на зеленую стрелку на панели управления Visual Studio (вы можете увидеть ее на рисунке 12-4 рядом с названием браузера, который будет использоваться для отображения приложения, в данном случае – Internet Explorer).

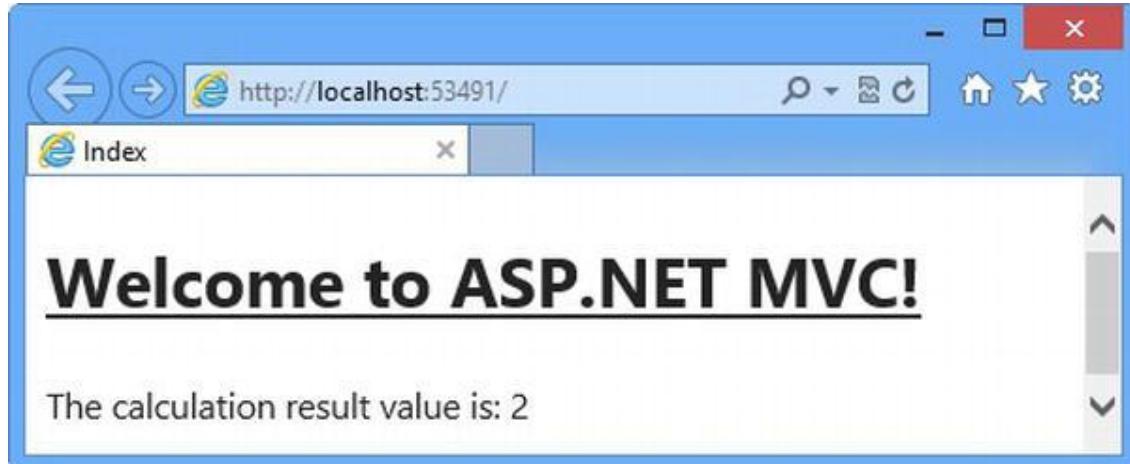
Если при запуске отладчика атрибут debug в файле Web.config установлен на false, то Visual Studio отобразит диалоговое окно, показанное на рисунке 12-6. Выберите вариант, который позволяет Visual Studio редактировать файл Web.config, и нажмите кнопку OK, и отладчик запустится.

**Рисунок 12-6:** Диалоговое окно, которое отображает Visual Studio, если в файле Web.config отключена отладка



Тогда ваше приложение запустится и отобразится в новом окне браузера, как показано на рисунке 12-7.

**Рисунок 12-7:** Запуск отладчика



Отладчик будет работать с приложением, но вы не заметите никакой разницы, пока отладчик не дойдет до точки остановки (мы объясняем, что это значит, в следующем разделе). Чтобы остановить отладчик, выберите `Stop Debugging` из Visual Studio меню `Debug` или закройте окно браузера.

## Остановка отладчика Visual Studio

Приложение, запущенное с отладчиком, будет вести себя нормально, пока не произойдет остановка, и в этот момент выполнение приложения прекращается и управление передается отладчику. И теперь вы можете проверять и контролировать состояние приложения. Остановки происходят по двум основным причинам: при достижении точки остановки (брекпойнта) или когда возникает необработанное исключение. Вы увидите примеры того и другого в следующих разделах.

### *Совет*

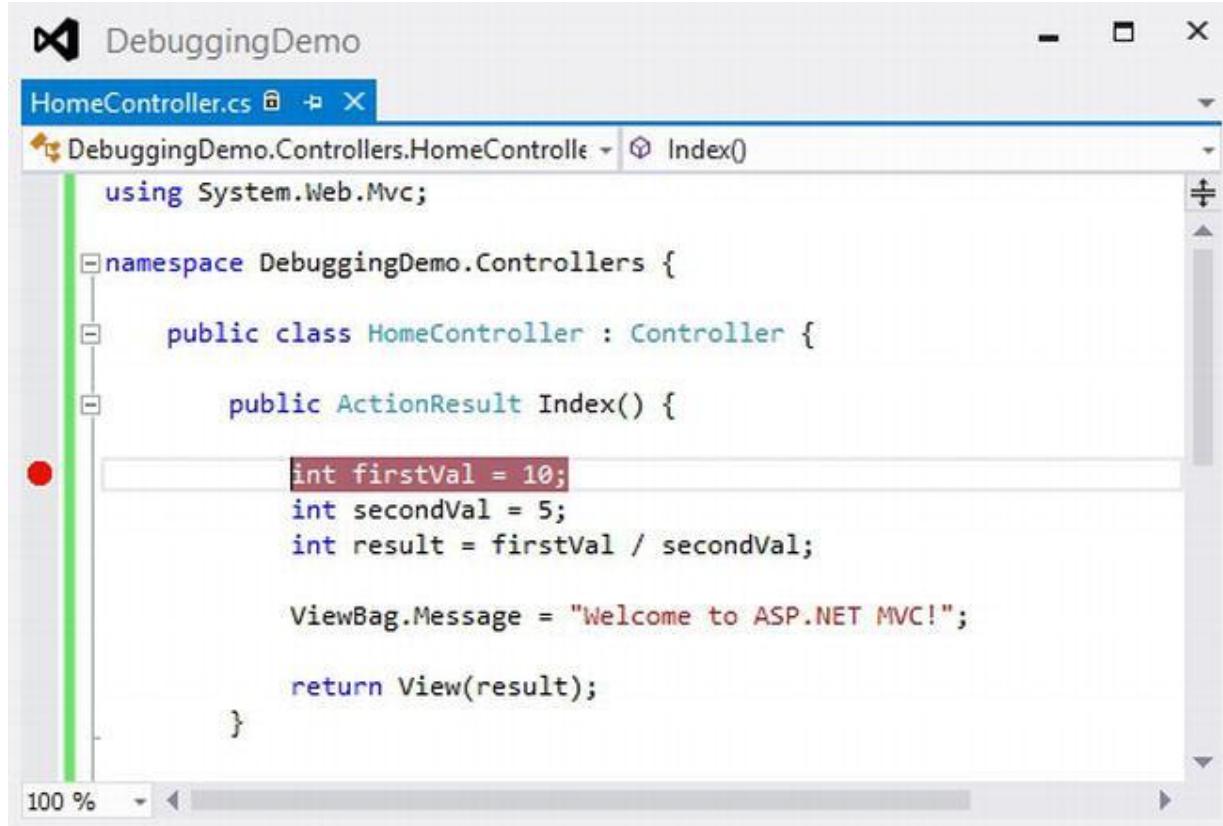
*Вы можете вручную остановить отладчик в любое время, выбрав `Break All` из Visual Studio меню `Debug`, в то время как работает отладчик.*

## Использование брекпойнтов

*Брекпойнт* – это инструкция, которая говорит отладчику остановить выполнение приложения и передать управление в руки программиста. В этот момент можно проверить состояние приложения, посмотреть, что происходит, и, при необходимости, возобновить выполнение приложения.

Чтобы создать брекпойнт, щелкните правой кнопкой мыши по выражению из кода и выберите из контекстного меню `Breakpoint` → `Insert Breakpoint`. Для примера поставьте брекпойнт напротив первого выражения метода действия `Index` контроллера `Home`, и вы увидите красную точку на поле текстового редактора, как показано на рисунке 12-8.

Рисунок 12-8: Использование брекпойнта для первого выражения метода действия Index



The screenshot shows the Visual Studio code editor with the file `HomeController.cs` open. The title bar says "DebuggingDemo". The code is as follows:

```
using System.Web.Mvc;

namespace DebuggingDemo.Controllers {
    public class HomeController : Controller {
        public ActionResult Index() {
            int firstVal = 10;
            int secondVal = 5;
            int result = firstVal / secondVal;

            ViewBag.Message = "Welcome to ASP.NET MVC!";

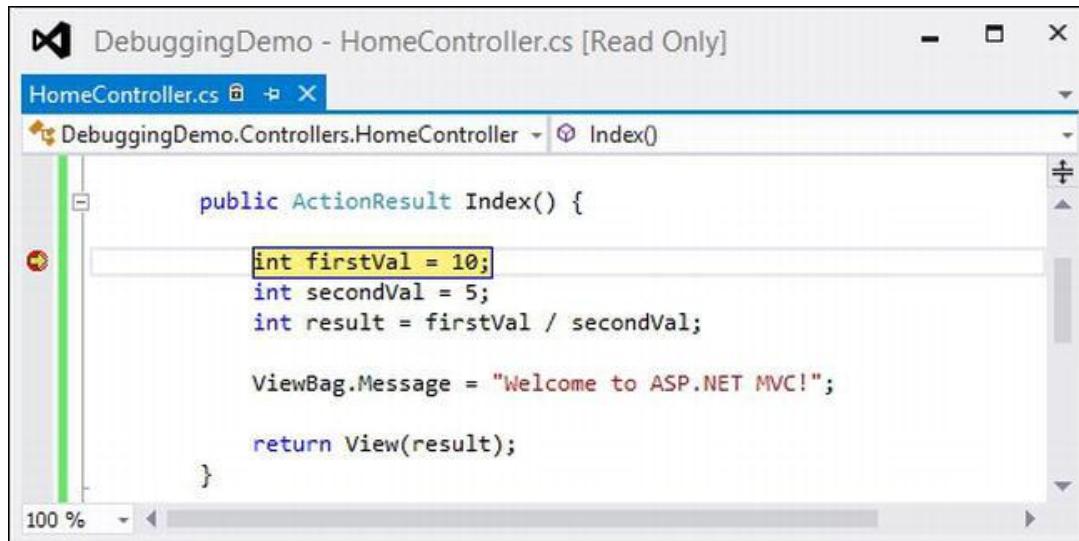
            return View(result);
        }
    }
}
```

A red circular breakpoint icon is positioned on the left margin next to the opening brace of the `Index()` method. The line `int firstVal = 10;` is highlighted with a red rectangle.

Чтобы увидеть результат от использования брекпойнта, запустите отладчик, выбрав Start Debugging из Visual Studio меню Debug. Приложение будет работать, пока не достигнет выражения, где был поставлен брекпойнт. Тогда отладчик остановится, приложение перестанет выполняться, и управление над ним перейдет к вам.

Visual Studio выделяет точки, когда было остановлено выполнение, желтым цветом, как показано на рисунке 12-9.

Рисунок 12-9: Нахождение брекпойнта



The screenshot shows the Visual Studio code editor with the file `HomeController.cs` open. The title bar says "DebuggingDemo - HomeController.cs [Read Only]". The code is the same as in Figure 12-8. A yellow circular execution point icon is on the left margin next to the opening brace of the `Index()` method. The line `int firstVal = 10;` is highlighted with a yellow rectangle.

### Примечание

Брекпойнт ловится только тогда, когда выполняется выражение, на которое он завязан. В нашем примере брекпойнт был достигнут сразу же, как только мы запустили приложение, потому что он находится внутри метода действия, который вызывается при запросе URL по умолчанию. Если вы поместите брекпойнт внутри другого метода действия, вы должны использовать браузер для запроса URL, связанного с этим методом.

Когда вы получаете управление над выполнением приложения, вы можете перейти к следующему выражению, выполнить операции в других методах и вообще исследовать состояние вашего приложения. Вы можете сделать это с помощью кнопки на панели инструментов или используя Debug меню Visual Studio.

Кроме того, чтобы обеспечить вам контроль над выполнением приложения, Visual Studio предоставляет много полезной информации о состоянии приложения: и этой информации так много, что она может занять все место в данной книге, поэтому мы опишем только самую основную.

### Просмотр значений данных в редакторе кода

Наиболее распространенное использование брекпойнтов заключается в том, чтобы попытаться разыскать ошибки в коде. Перед тем как исправить ошибку, вы должны выяснить, что же происходит, и одна из самых полезных функций, которые предоставляет Visual Studio, заключается в возможности просматривать и контролировать значения переменных прямо в редакторе кода.

В качестве примера запустите приложение, используя дебаггер, и ждите, пока не будет достигнут брекпойнт, который мы добавили в предыдущем разделе. Когда отладчик остановится, переместите указатель мыши на выражение, которое определяет переменную `result`. Вы увидите небольшое всплывающее окошко, которое показывает текущее значение, что видно на рисунке 12-10. На самом деле окошко не совсем такое, на рисунке мы показали его увеличенную версию.

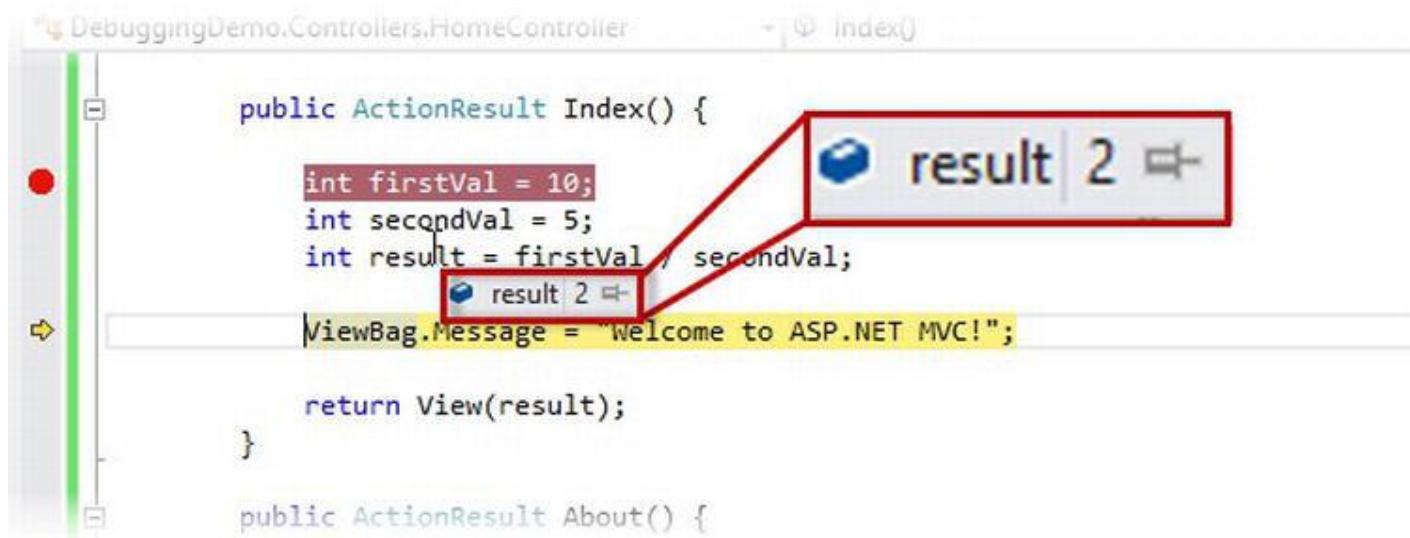
Рисунок 12-10: Отображение значения переменной в редакторе кода Visual Studio



Выполнение выражений в методе действия `Index` не достигло точки, где значения было присвоено переменной `result`, поэтому Visual Studio показывает нам значение по умолчанию, которое равно 0 типа `int`. Выберите пункт Step Over (или нажмите F10) в меню Visual Studio Debug, чтобы перейти к выполнению выражения, которое определяет свойство `ViewBag.Message`, и снова наведите курсор на

переменную `result`. Мы выполнили выражение, которое присваивает значение переменной `result`, и вы можете увидеть результат на рисунке 12-11.

Рисунок 12-11: Переменной присвоено значение



Мы часто используем эту функцию, когда начинаем процесс поиска ошибки, потому что благодаря этой возможности вы сразу понимаете, что происходит внутри вашего приложения. Это особенно полезно для выявления значений `null`, которые указывают, что переменной не было присвоено значение (и это причина многих ошибок, как мы знаем по опыту).

Вы видите, что справа от значения в окошке есть значок. Если вы нажмете на этот значок, всплывающее окошко покажет, когда значение переменной изменится: это позволяет контролировать одну или несколько переменных и видеть, когда они меняются и каковы их новые значения.

### Просмотр состояния приложения в окне отладчика

Visual Studio предоставляет несколько различных окон, которые можно использовать, чтобы получить информацию о вашем приложении, в то время как исполнение было приостановлено после брекпойнта. Полный список доступных окон отображается в `Debug` меню `Windows`, но двумя из наиболее полезных являются окна `Locals` и `Call Stack`. Окно `Locals` автоматически отображает значения всех переменных в текущей области, как показано на рисунке 12-12. Это дает вам возможность быстро просмотреть все переменные, которые вам интересны.

Рисунок 12-12: Окно Locals

The Locals window displays the following variable information:

Name	Value	Type
this	{DebuggingDemo.Controllers.HomeController}	DebuggingDemo.Controllers.HomeController
firstVal	10	int
secondVal	5	int
result	2	int

Переменные, значения которых были изменены ранее выполненным оператором, отображаются красным цветом: на рисунке значение переменной `result` красного цвета, потому что мы только что выполнили выражение, которое присваивает значение.

Набор переменных, показанных в окне `Locals`, меняется, когда вы перемещаетесь по приложению, но если вы хотите глобально отслеживать переменную, то щелкните правой кнопкой мыши по одному из элементов, показанных в окне `Locals`, и выберите опцию `Add Watch`. Элементы в окне `Watch` не будут меняться, когда вы выполняете выражения в приложении.

В окне `Call Stack` показана последовательность вызовов, которые привели к выполнению определенного оператора. Это может быть очень полезно, если вы пытаетесь понять странное поведение приложения, потому что вы можете просмотреть стек вызовов и определить обстоятельства, которые привели к брекпойнту. (Мы не показали вам окно `Call Stack` на рисунке, потому что наше приложение довольно простое и ему не хватает глубины стека, чтобы предоставить полезную информацию. Но мы рекомендуем вам изучить это и другие окна Visual Studio, чтобы понять, какую информацию способен предоставить отладчик).

#### *Совет*

*Вы можете добавлять брекпойнты в представления. Это может быть очень полезно, например, для проверки значений свойств модели представления. Вы добавляете брекпойнт в представление так же, как мы только что делали: щелкните правой кнопкой мыши по нужному выражению Razor и выберите Breakpoint -> Insert Breakpoint.*

## Остановка на исключении

Необработанные исключения являются частью процесса разработки. Одна из причин того, что мы создаем много модульных и интеграционных тестов в наших проектах, заключается в сведении к минимуму вероятности того, что такое исключение произойдет в развернутой версии приложения. Отладчик Visual Studio остановится автоматически, если увидит необработанное исключение.

#### *Примечание*

*Только необработанные исключения вызывают остановку отладчика. Исключение становится обработанным, если вы ловите и обрабатываете его в блоке try...catch. Обрабатываемые исключения являются полезным инструментом программирования. Они используются, чтобы представить сценарий, при котором метод не в состоянии выполнить свою задачу и должен сказать об этом. Необработанные исключения плохие, потому что они представляют неожиданные условия, которые мы не смогли предугадать (и потому что они приводят пользователя на страницу с ошибкой).*

Чтобы показать остановку на исключении, мы добавили небольшое изменение в метод действия `Index`, как показано в листинге 12-5.

#### **Листинг 12-5:** Добавление выражения, которое вызовет исключение

```
using System.Web.Mvc;  
namespace DebuggingDemo.Controllers
```

```

{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            int firstVal = 10;
            int secondVal = 0;
            int result = firstVal / secondVal;
            ViewBag.Message = "Welcome to ASP.NET MVC!";
            return View(result);
        }
    }
}

```

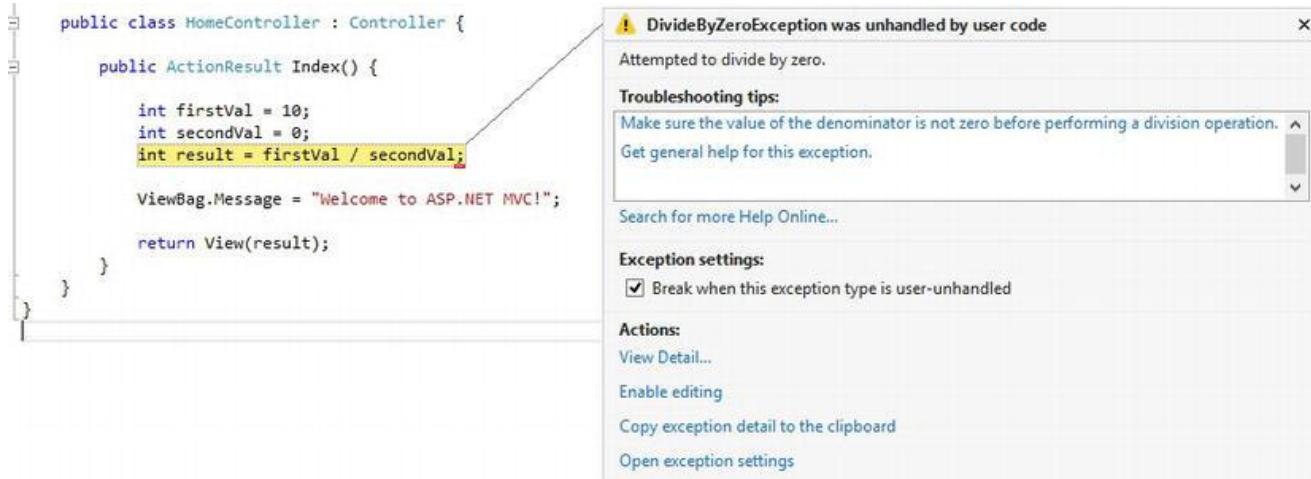
Мы изменили значение переменной `secondVal` на 0, что приведет к возникновению исключения в выражении, которое делит `firstVal` на `secondVal`.

### Примечание

Мы также удалили точку останова из метода действия `Index`, щелкнув правой кнопкой мыши по значку брекпойнта на поле и выбрав `Delete Breakpoint` из всплывающего меню.

При запуске отладчика приложение будет работать, пока не появится исключение: и в этот момент возникнет всплывающее окно, описывающее исключение, как показано на рисунке 12-13.

**Рисунок 12-13:** Всплывающее окно, описывающее исключение



Всплывающее окно дает вам информацию об исключении. Когда отладчик остановится на исключении, вы можете проверить состояние приложения так же, как и при использовании брекпойнта.

### Использование Edit and Continue

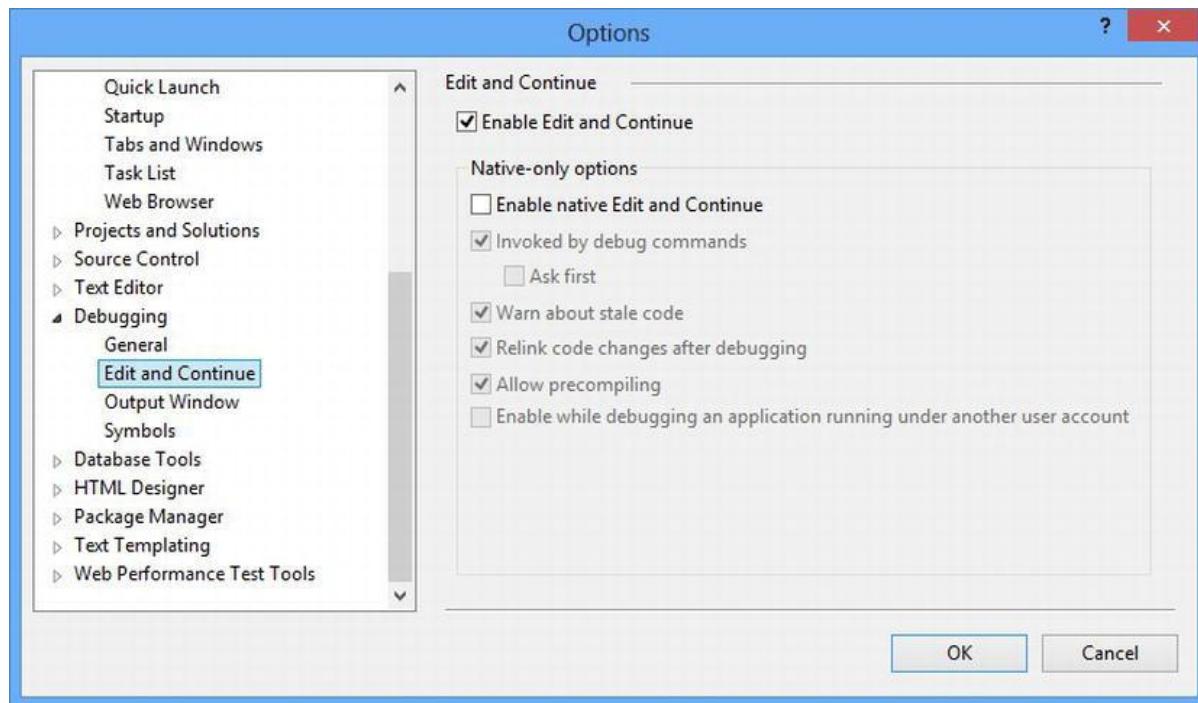
Интересная возможность отладки Visual Studio называется *Edit and Continue*. Когда отладчик остановится, вы можете отредактировать код, а затем продолжить отладку. Visual Studio перекомпилирует приложение и воссоздаст состояние приложения, каким оно было на момент, когда остановился отладчик.

## Включить Edit and Continue

Нам нужно включить Edit and Continue в двух местах:

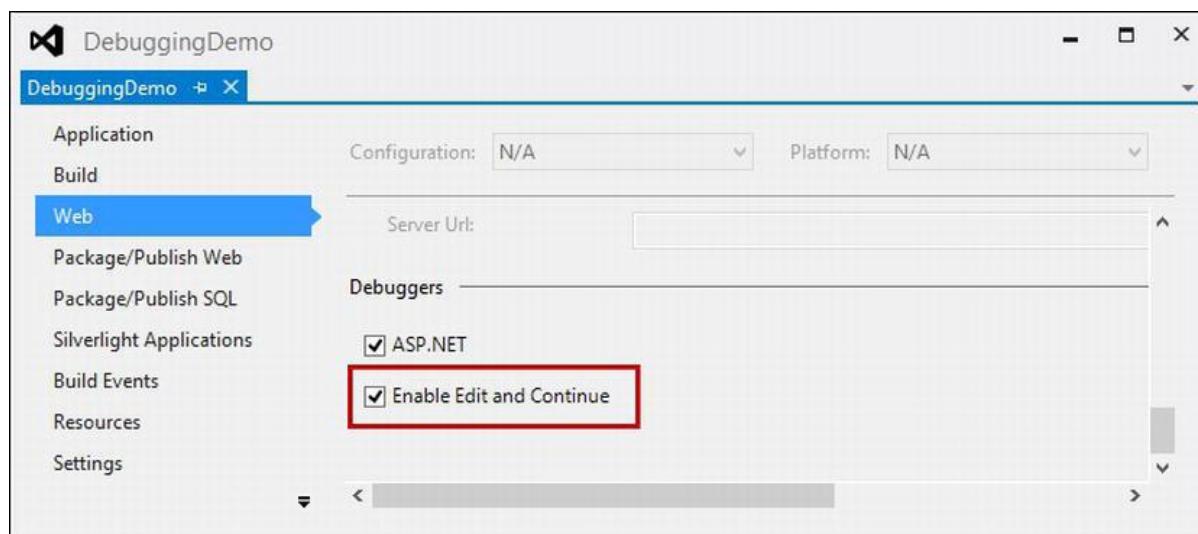
- В разделе Edit and Continue меню Debugging (выберите Options из меню Tools Visual Studio); убедитесь, что на Enable Edit and Continue поставлена галочка, как показано на рисунке 12-14.

Рисунок 12-14: Включение Edit and Continue в диалоговом окне Options



- В свойствах проекта (выберите DebuggingDemo Properties из Visual Studio меню Project); щелкните раздел Web и убедитесь, что на Enable Edit and Continue поставлена галочка, как показано на рисунке 12-15.

Рисунок 12-15: Включение Edit and Continue в свойствах проекта



## Изменение проекта

Возможность Edit and Continue несколько придергчива. Есть несколько условий, при которых она не может работать. Одно из таких условий присутствует в методе действия Index класса HomeController: использование динамических объектов. Чтобы обойти это, мы закомментировали строку, которая использует ViewBag в классе HomeController.cs, как показано в листинге 12-6.

### Листинг 12-6: Удаление вызова ViewBag из метода Index

```
using System.Web.Mvc;
namespace DebuggingDemo.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            int firstVal = 10;
            int secondVal = 0;
            int result = firstVal / secondVal;
            // Это выражение закомментировано
            //ViewBag.Message = "Welcome to ASP.NET MVC!";
            return View(result);
        }
    }
}
```

Мы должны сделать соответствующее изменение в представлении Index.cshtml, как показано в листинге 12-7.

### Листинг 12-7: Удаление вызова ViewBag из представления

```
@model int
 @{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link href="~/Content/Site.css" rel="stylesheet" type="text/css" />
    <title>Index</title>
</head>
<body>
    <!-- Этот элемент закомментирован -->
    <!--<h2 class="message">@ ViewData["Message"]</h2>-->
    <p>
        The calculation result value is: @Model
    </p>
</body>
</html>
```

## Редактирование и продолжение

Мы готовы к демонстрации возможности Edit and Continue. Выберите Start Debugging из Visual Studio меню Debug. Приложение запустится с отладчиком и будет работать до тех пор, пока не достигнет линии, где мы выполняем несложное вычисление в методе Index. Значение второго параметра равно нулю, что приводит к исключению. В этот момент отладчик останавливается, и появляется всплывающее окно с подсказкой (как показано на рисунке 12-13).

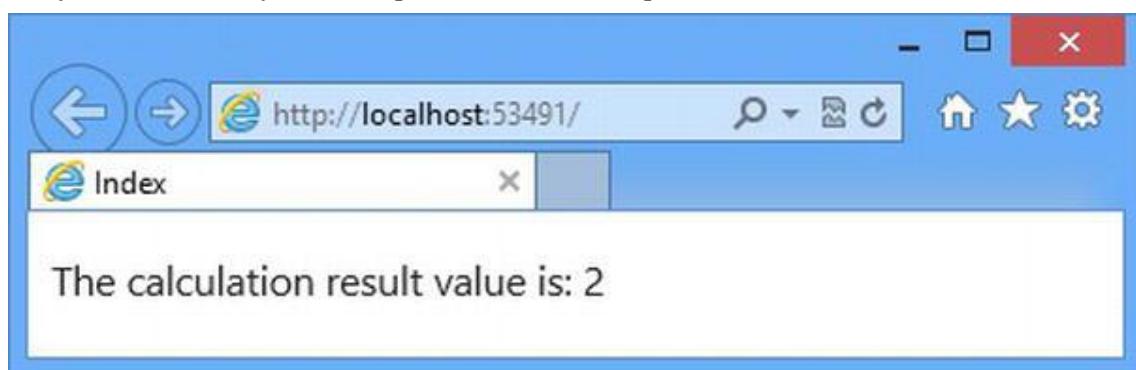
Нажмите ссылку Enable editing в окне с подсказкой. В редакторе кода наведите курсор мыши на переменную `secondVal` и нажмите на значение в появившемся всплывающем окне. Введите 5 в качестве нового значения, как показано на рисунке 12-16.

**Рисунок 12-16:** Изменение значения переменной

```
public ActionResult Index() {  
    int firstVal = 10;  
    int secondVal = 0;  
    int result = firstVal / secondVal; // This statement has been commented out  
    //ViewBag.Message = "Welcome to ASP.NET MVC!"  
  
    return View(result);
```

Теперь выберите Continue из Visual Studio меню Debug, чтобы возобновить выполнение приложения. Новое значение, которое вы присвоили переменной, используется для генерации значения переменной `result`, как показано на рисунке 12-17.

**Рисунок 12-17:** Результат исправления ошибки при помощи Edit and Continue



Найдите минутку, чтобы понять то, что произошло здесь. Мы запустили программу с ошибкой: это была попытка разделить значение на ноль. Отладчик обнаружил исключение и остановил выполнение программы. Мы отредактировали код, чтобы исправить ошибку, заменив значение переменной на 5. Затем мы сказали отладчику продолжить выполнение программы.

И в этот момент Visual Studio перекомпилировала наше приложение так, чтобы изменения были включены в процесс сборки, заново запустила выполнение программы, воссоздала состояние, которые привело к исключению, а затем продолжила работу, как обычно. В браузере отобразился результат, к которому привели наши изменения.

Без `Edit and Continue` нам нужно было бы остановить приложение, сделать изменения, скомпилировать приложение и перезапустить отладчик. Затем нам нужно было бы использовать браузер, чтобы повторить шаги, которые мы сделали до момента, когда остановился отладчик. Функция `Edit and Continue` является достаточно важной. Исправление сложных ошибок может потребовать повторения множества шагов по всему приложению, а возможность протестировать потенциальные исправления без необходимости повторять эти шаги снова и снова может сильно сэкономить время программиста.

# Резюме

В этой главе мы показали вам структуру MVC проекта Visual Studio и то, как различные части сочетаются друг с другом. Мы также затронули одну из самых важных характеристик MVC фреймворка: соглашений. Это темы, к которым мы будем возвращаться снова и снова в последующих главах, когда мы будем глубже исследовать, как работает MVC фреймворк.

# Маршрутизация

До введения MVC ASP.NET предполагал, что существует прямая связь между запрашиваемым URL-адресом и файлами на жестком диске сервера. Работы сервера заключалась в том, чтобы получить запрос от браузера и доставить выходные данные из соответствующего файла, а именно:

Запрашиваемый URL	Соответствующий файл
http://mysite.com/default.aspx	e:\webroot\default.aspx
http://mysite.com/admin/login.aspx	e:\webroot\admin\login.aspx
http://mysite.com/articles/AnnualReview	Файл не найден. Ошибка 404

Этот подход очень хорош для Web Forms, где каждая ASPX страница является и файлом, и ответом на запрос. Но это не имеет смысла для MVC приложения, где запросы обрабатываются методами действия в классах контроллеров и где нет однозначного соответствия с файлами на диске.

Для обработки MVC URL платформа ASP.NET использует систему маршрутизации (роутинга). В этой главе мы покажем вам, как настроить и использовать систему маршрутизации для создания мощной и гибкой обработки URL для ваших проектов. Вы увидите, что система маршрутизации позволяет создавать любые желаемые структуры URL и выражать их в ясной и сжатой форме.

Система маршрутизации имеет две функции:

- Изучить *входящий URL* и выяснить, для какого контроллера и действия предназначен запрос. Как и следовало ожидать, мы хотим, чтобы система маршрутизации делала именно это, когда мы получаем запрос клиента.
- Создать *исходящих URL*. Это URL-адреса, которые появляются в HTML документе, показанном нашим представлением, так чтобы было вызвано определенное действие, когда пользователь кликает по ссылке (в этот момент мы снова получаем входящий URL).

В этой главе мы сосредоточимся на определении роутов и их использование для обработки входящих запросов, так чтобы пользователь мог достичь контроллеров и действий. Затем, в следующей главе, мы покажем вам, как использовать те же самые маршруты для генерации исходящих URL, которые будут вам нужны для представлений, а также покажем, как настроить систему маршрутизации и использовать ее для *области* (*areas*).

## Создание проекта для примера

Для демонстрации системы маршрутизации нам нужен проект, к которому мы можем добавить роуты. Мы создали новое MVC приложение, используя шаблон `Basic`, и назвали проект `UrlsAndRoutes`.

### *Совет*

*Мы включили в эту главу различные модульные тесты, и если вы хотите воссоздать их, то вам нужно будет выбрать опцию `Create a unit test project`, когда вы выбираете шаблон `Basic`, и использовать NuGet, чтобы добавить `Moq` в проект юнит тестирования.*

Для демонстрации роутов мы собираемся добавить некоторые простые контроллеры в приложение. В этой главе мы хотим только показать способ, которым интерпретируются URL, чтобы вызвать методы действия, поэтому используемые модели представления являются строковыми значениями

в ViewBag, который сообщает имена контроллера и метода действия. Во-первых, создайте контроллер Home и установите его содержание, чтобы оно совпадало с тем, что в листинге 13-1.

#### Листинг 13-1: Содержание контроллера Home

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
namespaceUrlsAndRoutes.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewBag.Controller = "Home";
            ViewBag.Action = "Index";
            return View("ActionName");
        }
    }
}
```

Создайте контроллер Customer и с содержанием, как в листинге 13-2.

#### Листинг 13-2: Содержание контроллера Customer

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
namespaceUrlsAndRoutes.Controllers
{
    public class CustomerController : Controller
    {
        public ActionResult Index()
        {
            ViewBag.Controller = "Customer";
            ViewBag.Action = "Index";
            return View("ActionName");
        }
        public ActionResult List()
        {
            ViewBag.Controller = "Customer";
            ViewBag.Action = "List";
            return View("ActionName");
        }
    }
}
```

Создайте второй контроллер Admin, как в листинге 13-3.

#### Листинг 13-3: Содержание контроллера Admin

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
namespaceUrlsAndRoutes.Controllers
{
    public class AdminController : Controller
    {
```

```

public ActionResult Index()
{
    ViewBag.Controller = "Admin";
    ViewBag.Action = "Index";
    return View("ActionName");
}
}

```

Мы определили представление ActionName во всех методах действия этих контроллеров, что позволяет нам определить одно представление и использовать его во всем приложении. Добавьте новое представление с именем ActionName.cshtml в папку /Views/Shared и установите его содержание, чтобы оно соответствовало тому, что показано в листинге 13-4.

**Листинг 13-4:** Содержание представления ActionName.cshtml

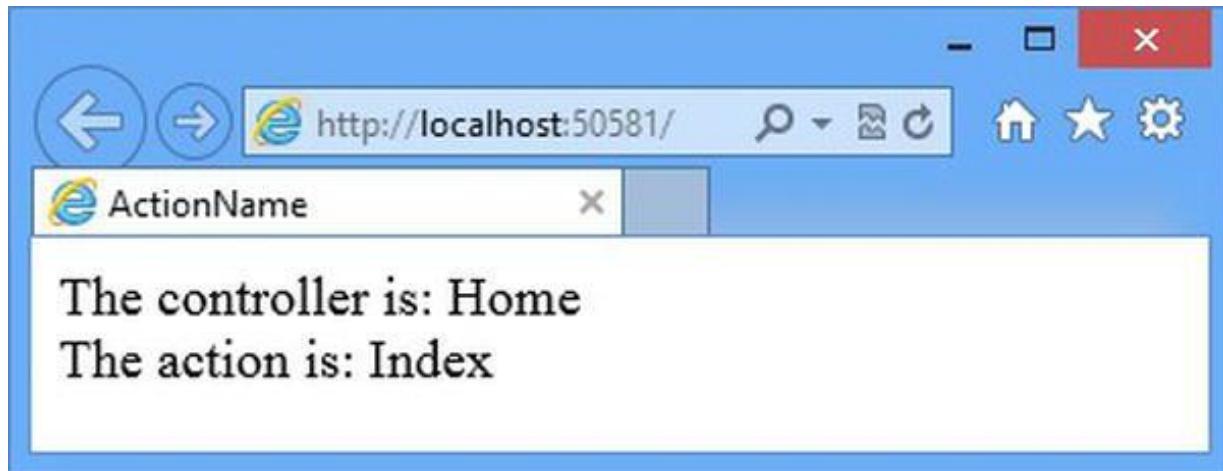
```

@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>ActionName</title>
</head>
<body>
    <div>The controller is: @ViewBag.Controller</div>
    <div>The action is: @ViewBag.Action</div>
</body>
</html>

```

Если вы запустите приложение, вы увидите содержимое, как на рисунке 13-1.

**Рисунок 13-1:** Запуск приложения



## Введение в URL паттерны

Система маршрутизации использует набор *роутов*. Эти маршруты совместно составляют *URL схему* приложения, являющуюся набором URL-адресов, которые ваше приложение будет распознавать и на которые будет реагировать.

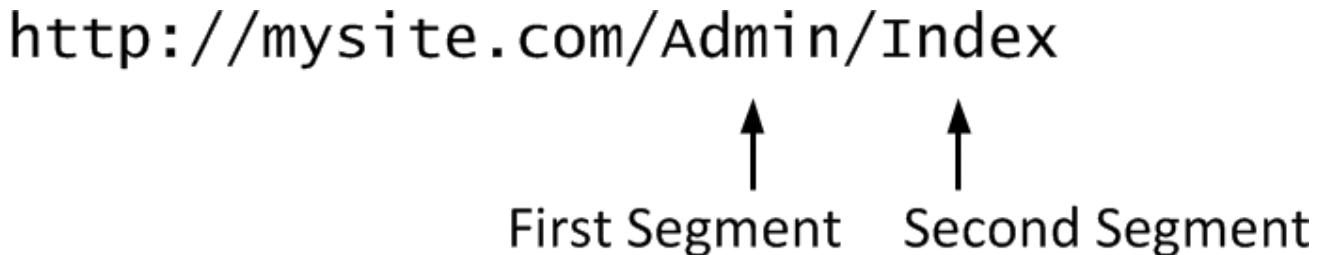
Нам не нужно вручную набирать отдельные URL, которые мы будем поддерживать. Вместо этого каждый роут содержит *URL паттерн*, который сравнивается с входящим URL. Если паттерн

соответствует URL, тогда его использует система маршрутизации для обработки этого URL. Давайте начнем с URL для нашего приложения:

`http://mysite.com/Admin/Index`

URL-адреса могут быть разбиты на *сегменты*. Это части URL, исключая хост и строку запроса, разделенные символом «/». В URL из примера есть два сегмента, как показано на рисунке 13-2.

**Рисунок 13-2:** Сегменты URL из примера



Первый сегмент содержит слово `Admin`, а второй сегмент содержит слово `Index`. Для нас очевидно, что первый сегмент относится к контроллеру, а второй сегмент – к действию. Но, конечно, мы должны выразить эти отношения таким образом, чтобы система маршрутизации могла это понять. Вот URL паттерн, который делает это:

```
{controller}/{action}
```

При обработке входящего запроса работа системы маршрутизации заключается в том, чтобы запрошенный URL подходил паттерну, а также в извлечении значения из URL для *переменных сегмента*, определенных в шаблоне. Сегментные переменные выражаются при помощи фигурных скобок (`{ }` ). Например, в паттерне есть две сегментные переменные с именами `controller` и `action`, и поэтому значение сегментной переменной `controller` будет `Admin`, а значение переменной сегмента `action` будет `Index`.

Мы указываем на соответствие *a* паттерну, потому что в MVC приложении, как правило, есть несколько роутов, и система маршрутизации сравнивает входящий URL с URL паттерном каждого роута, пока не найдет соответствия.

#### Примечание

*Система маршрутизации не имеет специальных знаний о контроллерах и действиях. Она просто извлекает значения для сегментных переменных и передает их по каналу запросов. А затем в канале обработки запросов, если запрос достигает MVC фреймворка, это значение присваивается переменным controller и action. Именно поэтому система маршрутизации может быть использована с Web Forms и Web API (Web API рассматривается в главе 25).*

По умолчанию URL паттерн будет соответствовать любому URL, который имеет правильное количество сегментов. Например, паттерн `{controller}/{action}` будет соответствовать любому URL, который имеет два сегмента, как показано в таблице 13-1.

Таблица 13-1: Подходящие URL

URL запроса	Переменные сегмента
http://mysite.com/Admin/Index	controller = Admin, action = Index
http://mysite.com/Index/Admin	controller = Index, action = Admin
http://mysite.com/Apples/Oranges	controller = Apples, action = Oranges
http://mysite.com/Admin	Нет соответствий: слишком мало сегментов
http://mysite.com/Admin/Index/Soccer	Нет соответствий: слишком много сегментов

В таблице 13-1 выделены два ключевых вида поведения URL паттернов:

- URL паттерны *консервативны*: они будут соответствовать только тем URL, которые имеют одинаковое число сегментов, что и нужный паттерн. Вы можете увидеть это в четвертом и пятом примерах в таблице.
- URL паттерны *либеральны*. Если URL имеет правильное количество сегментов, паттерн извлечет значение для сегментной переменной, каким бы оно ни было.

Это поведение по умолчанию, которые являются ключом к пониманию того, как работают URL паттерны. Мы покажем вам, как изменить настройки по умолчанию, далее в этой главе.

Как мы уже упоминали, система маршрутизации ничего не знает об MVC приложении, поэтому URL паттерн найдет соответствие, даже если нет контроллера и действия, которые подходят значениям, извлеченным из URL. Мы показали это во втором примере в таблице 13-1. Мы поменяли сегменты URL Admin и Index, поэтому и значения, извлекаемые из URL, также меняются местами, хотя в примере нет никакого контроллера Index.

## Создание и регистрация простого роута

Когда у вас есть URL паттерн, вы можете использовать его для определения роута. Роуты находятся в файле `RouteConfig.cs`, который находится в папке проекта `App_Start`. Вы можете увидеть исходное содержание, которое Visual Studio определяет для этого файла, в листинге 13-5.

Листинг 13-5: Содержание по умолчанию файла `RouteConfig.cs`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
namespaceUrlsAndRoutes
{
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
            routes.MapRoute(

```

```
        name: "Default",
        url: "{controller}/{action}/{id}",
        defaults: new
        {
            controller = "Home",
            action = "Index",
            id = UrlParameter.Optional
        }
    );
}
}
```

Статический метод `RegisterRoutes`, который определен в файле `RouteConfig.cs`, вызывается из файла `Global.asax.cs`, который устанавливает некоторые из основных функциональных возможностей MVC при запуске приложения. Вы можете увидеть содержимое по умолчанию файла `Global.asax.cs` в листинге 13-6, и мы выделили вызов метода `RouteConfig.RegisterRoutes`, который сделан из метода `Application_Start`.

### Листинг 13-6: Содержание по умолчанию Global.asax.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Http;
using System.Web.Mvc;
using System.Web.Optimization;
using System.Web.Routing;
namespaceUrlsAndRoutes
{
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();
            WebApiConfig.Register(GlobalConfiguration.Configuration);
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);
        }
    }
}
```

Метод `Application_Start` вызывается базовой платформой ASP.NET, когда MVC приложение запускается в первый раз, что приводит к вызову метода `RouteConfig.RegisterRoutes`. Параметром этого метода является значение статического свойства `RouteTable.Routes`, которое является экземпляром класса `RouteCollection` (мы опишем его далее).

Совет

*Другие вызовы, сделанные методом Application\_Start, рассматриваются в других главах. Мы описываем метод AreaRegistration.RegisterAllAreas в главе 14, метод WebApiConfig.Register в главе 25, метод FilterConfig.RegisterGlobalFilters в главе 16 и метод BundleConfig.RegisterBundles в главе 24.*

В листинге 13-7 показано, как создать роут, используя пример URL паттерна из предыдущего раздела, в методе `RegisterRoutes` файла `RouteConfig.cs` (Мы удалили из метода все другие выражения, чтобы можно было сосредоточиться на примере).

**Листинг 13-7:** Регистрация роута

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
namespaceUrlsAndRoutes
{
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            Route myRoute = new Route("{controller}/{action}", new MvcRouteHandler());
            routes.Add("MyRoute", myRoute);
        }
    }
}
```

Мы создаем новый роут, используя наш URL паттерн как параметр конструктора, который мы выражаем в виде строки. Мы также передаем конструктору экземпляр `MvcRouteHandler`. Различные ASP.NET технологии предлагают различные классы для адаптации правил маршрутизации, и этот класс мы используем для ASP.NET MVC приложений. Как только мы создали роут, мы добавляем его в объект `RouteCollection`, используя метод `Add`, передав ему имя, под которым будет известен роут, и созданный нами роут.

*Совет*

***Именование маршрутов не является обязательным, и есть мнение, что таким образом мы жертвуем чистым разделением понятий, которое в ином случае может предоставить роутинг. Мы не сильно переживаем по поводу именования, но мы объясним, почему это может быть проблемой, в разделе "Создание URL из конкретного роута" далее в этой главе.***

---

Более удобным способом регистрации роутов является использование метода `MapRoute`, определенного в классе `RouteCollection`. Листинг 13-8 показывает, как мы можем использовать этот метод, чтобы зарегистрировать роут. Результат этого такой же, как и в предыдущем примере, но синтаксис тут чище.

**Листинг 13-8:** Регистрация роута при помощи метода `MapRoute`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
namespaceUrlsAndRoutes
{
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
```

```

        routes.MapRoute("MyRoute", "{controller}/{action}");
    }
}

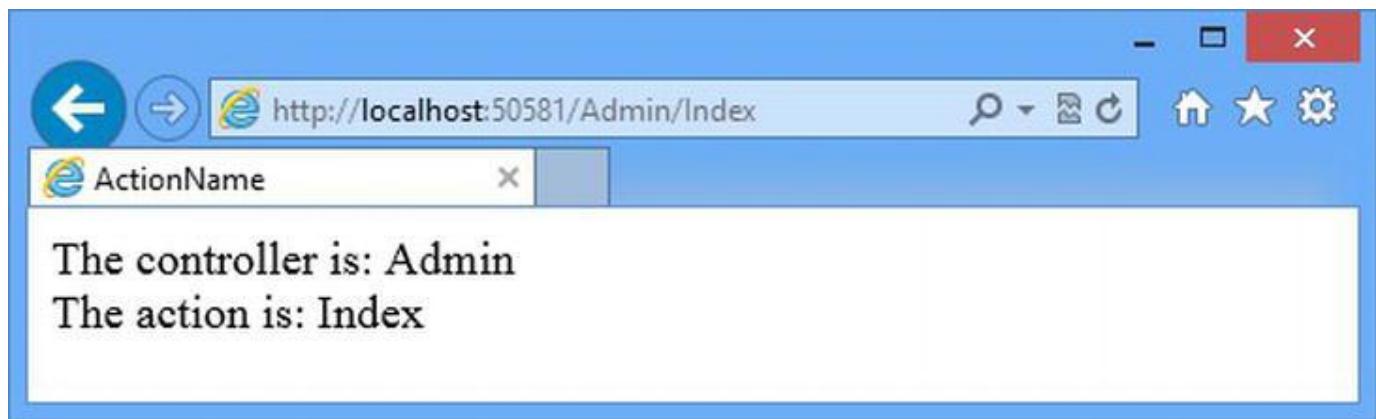
```

Такой подход является чуть более компактным, в основном потому, что мы не должны создавать экземпляр класса `MvcRouteHandler`. Метод `MapRoute` предназначен исключительно для использования с MVC приложениями. Приложения ASP.NET Web Forms могут использовать метод `MapPageRoute`, также определяемый в классе `RouteCollection`.

### Использование простого роута

Вы можете увидеть результат изменений, которые мы сделали в маршрутизации, запустив приложение. Когда браузер пытается перейти к корневому URL приложения, вы увидите сообщение об ошибке, но если перейти к роуту, который соответствует нашему паттерну `{controller}/{action}`, вы увидите результат, показанный на рисунке 13-3, где проиллюстрирован переход к `/Admin/Index`.

**Рисунок 13-3:** Навигация при помощи простого роута



Наш простой роут не говорит MVC фреймворку, как реагировать на запросы для корневого URL, и поддерживает только один очень конкретный URL паттерн. Мы сейчас временно сделали шаг назад от функционала, который Visual Studio добавляет в файл `RouteConfig.cs` при создании MVC проекта. Мы покажем вам, как создавать более сложные паттерны и роуты, в остальной части этой главы.

### Юнит тест: тестирование входящих URL

Мы рекомендуем вам проводить модульное тестирование роутов, чтобы убедиться, они обрабатывают входящие URL нужным образом, даже если вы решите не проводить модульное тестирование остальной части вашего приложения. URL схемы могут быть довольно сложными в больших приложениях, и поэтому могут преподнести вам сюрпризы и неожиданные результаты.

В предыдущих главах мы избегали создания общих вспомогательных методов для совместного использования тестов для того, чтобы каждый юнит тест был самодостаточным. Для этой главы мы используем другой подход. Легче всего сделать тестирование схемы маршрутизации для приложения, когда вы объединяете несколько тестов в одном методе, и это намного проще сделать благодаря некоторым вспомогательным методам.

Для проверки роутов нам нужно использовать мок-технологию для трех классов из MVC фреймворка: `HttpRequestBase`, `HttpContextBase` и `HttpResponseBase` (последний класс требуется

для проверки исходящих URL, которые мы рассмотрим в следующей главе). Вместе эти классы воссоздают достаточно инфраструктуры MVC для поддержки системы маршрутизации. Мы добавили новый файл юнит тестов `RouteTests.cs` в проект модульного тестирования, и нашим первым дополнением является вспомогательный метод, который создает mock-объекты `HttpContextBase`:

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System.Web;
using System.Web.Routing;
using Moq;
using System.Reflection;
namespaceUrlsAndRoutes.Tests
{
    [TestClass]
    public class RouteTests
    {
        private HttpContextBase CreateHttpContext(string targetUrl = null, string httpMethod = "GET")
        {
            // создать mock-запрос
            Mock<HttpRequestBase> mockRequest = new Mock<HttpRequestBase>();
            mockRequest.Setup(m => m.AppRelativeCurrentExecutionFilePath)
                .Returns(targetUrl);
            mockRequest.Setup(m => m.HttpMethod).Returns(httpMethod);

            // создать mock-response
            Mock<HttpResponseBase> mockResponse = new Mock<HttpResponseBase>();
            mockResponse.Setup(m => m.ApplyAppPathModifier(It.IsAny<string>()))
                .Returns<string>(s => s);

            // создать mock-контекст, используя запрос и ответ
            Mock<HttpContextBase> mockContext = new Mock<HttpContextBase>();
            mockContext.Setup(m => m.Request).Returns(mockRequest.Object);
            mockContext.Setup(m => m.Response).Returns(mockResponse.Object);
            // вернуть mock-контекст
            return mockContext.Object;
        }
    }
}
```

Здесь все проще, чем кажется. Мы раскрываем URL, который мы хотим проверить, используя свойство `AppRelativeCurrentExecutionFilePath` класса `HttpRequestBase`, и раскрываем `HttpRequestBase` благодаря свойству `Request` mock-класса `HttpContextBase`. Наш следующий вспомогательный метод позволяет нам протестировать роут:

```
...
private void TestRouteMatch(string url, string controller, string action,
    object routeProperties = null, string httpMethod = "GET")
{
    // Arrange
    RouteCollection routes = new RouteCollection();
    RouteConfig.RegisterRoutes(routes);
    // Act - обрабатывает роут
    RouteData result
        = routes.GetRouteData(CreateHttpContext(url, httpMethod));
    // Assert
    Assert.IsNotNull(result);
    Assert.IsTrue(TestIncomingRouteResult(result, controller,
        action, routeProperties));
}
...
```

Параметры этого метода позволяют задавать URL для тестирования, ожидаемые значения сегментных переменных controller и action, а также object, который содержит ожидаемые значения для любых дополнительных переменных, которые мы определили. Мы покажем вам, как создавать такие переменные, далее в этой главе. Мы также определили параметр для HTTP метода, которые мы объясним в разделе "Ограничение роутов".

Метод TestRouteMatch полагается на другой метод, TestIncomingRouteResult, чтобы сравнить результаты, полученные от системы маршрутизации, с ожидаемыми значениями сегментных переменных. Этот метод использует .NET рефлексию, поэтому мы можем использовать анонимный тип, чтобы выразить любые дополнительные переменные сегмента. Не волнуйтесь, если этот метод вам непонятен, так как он нужен, чтобы сделать тестирование более удобным, но не является обязательным требованием для понимания MVC. Вот метод TestIncomingRouteResult:

```
private bool TestIncomingRouteResult(RouteData routeResult,
    string controller, string action, object propertySet = null)
{
    Func<object, object, bool> valCompare = (v1, v2) => {
        return StringComparer.InvariantCultureIgnoreCase
            .Compare(v1, v2) == 0;
    };

    bool result = valCompare(routeResult.Values["controller"], controller)
        && valCompare(routeResult.Values["action"], action);
    if (propertySet != null) {
        PropertyInfo[] propInfo = propertySet.GetType().GetProperties();
        foreach (PropertyInfo pi in propInfo) {
            if (!routeResult.Values.ContainsKey(pi.Name)
                && valCompare(routeResult.Values[pi.Name],
                    pi.GetValue(propertySet, null))))
            {
                result = false;
                break;
            }
        }
    }
    return result;
}
```

Нам также нужен метод, чтобы проверить, что URL не работает. Как вы увидите, это может быть важной частью определения URL схемы.

```
...
private void TestRouteFail(string url)
{
    // Arrange
    RouteCollection routes = new RouteCollection();
    RouteConfig.RegisterRoutes(routes);
    // Act - обработка роута
    RouteData result = routes.GetRouteData(CreateHttpContext(url));
    // Assert
    Assert.IsTrue(result == null || result.Route == null);
}
...
```

TestRouteMatch и TestRouteFail содержат вызов метода Assert, который генерирует исключение, если утверждение не выполняется. Поскольку в C# исключения передаются вверх по стеку вызовов, мы можем создавать простые тестовые методы, которые проверяют набор URL-адресов и получают нужное нам тестовое поведение. Вот тестовый метод, проверяющий роут, который мы определили в листинге 13-8:

```

...
[TestMethod]
public void TestIncomingRoutes() {
    // проверка на URL, который мы надеемся получить
    TestRouteMatch("~/Admin/Index", "Admin", "Index");
    // проверка на то, что значения были получены из сегментов
    TestRouteMatch("~/One/Two", "One", "Two");
    // гарантия того, что слишком много или слишком мало сегментов не подходят
    TestRouteFail("~/Admin/Index/Segment");
    TestRouteFail("~/Admin");
}
...

```

Этот тест использует метод `TestRouteMatch` для проверки URL, который мы ожидаем. Он также проверяет URL в том же формате, чтобы убедиться, что значения `controller` и `action` получены надлежащим образом с помощью URL сегментов. Мы также используем метод `TestRouteFail`, чтобы убедиться, что наше приложение не будет принимать URL, которые имеют другое количество сегментов. При тестировании мы должны поставить перед URL тильду (~), потому что таким образом ASP.NET Framework представляет URL системе маршрутизации.

Обратите внимание, что нам не нужно в тестовых методах определять роуты. Это потому что мы загружаем их напрямую с помощью метода `RegisterRoutes` в классе `RouteConfig`.

## Определение значений по умолчанию

Причина того, что мы получили сообщение об ошибке, когда мы запросили URL по умолчанию для приложения, заключается в том, что он не соответствовал заданному роуту. URL по умолчанию выражается для системы маршрутизации как `~/`, и нет никаких сегментов в этой строке, которые могут подойти переменным `controller` и `action`, определенным нашей простой роутинговой схемой.

Мы объяснили ранее, что URL паттерны носят консервативный характер в том, что они будут соответствовать только URL с указанным количеством сегментов. Мы также сказали, что это поведение по умолчанию. Один из способов изменить это поведение – это использовать *значения по умолчанию*. Значение по умолчанию применяется тогда, когда URL не содержит сегмент, который может соответствовать значению. В листинге 13-9 показан пример роута, который содержит значение по умолчанию.

*Примечание*

*Начиная с этого момента, когда мы покажем вам новую конфигурацию маршрутизации. Мы работаем с измененным методом `RegisterRoutes` класса `RouteConfig`.*

---

**Листинг 13-9:** Добавление значения по умолчанию в роут

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
namespaceUrlsAndRoutes
{
    public class RouteConfig

```

```

    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.MapRoute("MyRoute", "{controller}/{action}",
                new { action = "Index" });
        }
    }
}

```

Значения по умолчанию поставляются как свойства в анонимном типе. В листинге 13-9 мы предоставили значение по умолчанию `Index` для переменной `action`. Этот роут будет соответствовать всем двухсегментным URL, как это было ранее. Например, если запрашивается URL `http://mydomain.com/Home/Index`, роут будет извлекать `Home` как значение для `controller` и `Index` в качестве значения для `action`.

Но теперь, когда мы указали значение по умолчанию для сегмента `action`, роут будет также соответствовать односегментному URL. При обработке односегментного URL система маршрутизации извлекает значение `controller` из одного URL сегмента и использует значение по умолчанию для переменной `action`. Таким образом, мы можем запросить URL `http://mydomain.com/Home` и вызвать метод действия `Index` контроллера `Home`.

Мы можем пойти дальше и определить URL, которые вообще не содержат сегментных переменных, полагаясь только на значения по умолчанию для идентификации действия и контроллера. Мы можем отобразить URL по умолчанию, используя значения по умолчанию для обоих, как показано в листинге 13-10.

**Листинг 13-10:** Добавление в роут значений по умолчанию для действия и контроллера

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
namespaceUrlsAndRoutes
{
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.MapRoute("MyRoute", "{controller}/{action}",
                new { controller = "Home", action = "Index" });
        }
    }
}

```

Поскольку мы предоставили значения по умолчанию для переменных `controller` и `action`, мы создали роут, который будет соответствовать URL, имеющим ноль, один или два сегмента, как показано в таблице 13-2.

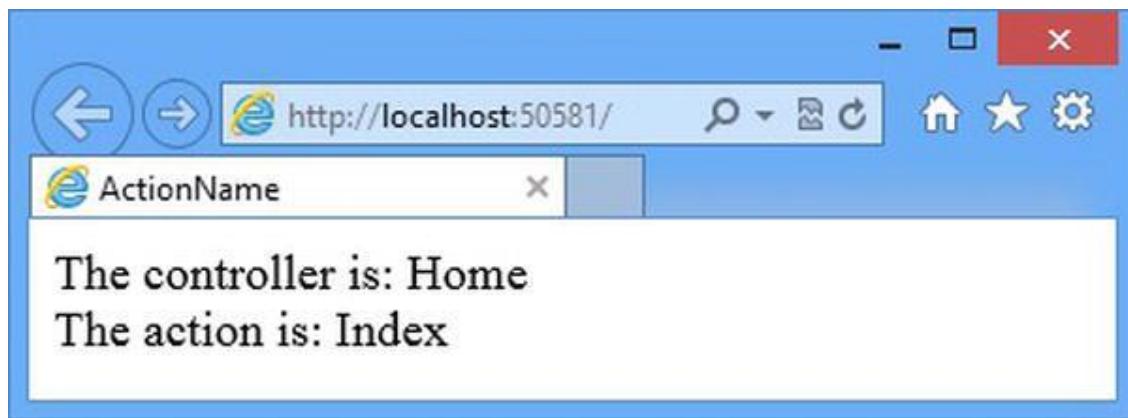
**Таблица 13-2:** Подходящие URL

Число сегментов	Пример	Соответствие
0	mydomain.com	controller = Home, action = Index

Число сегментов	Пример	Соответствие
1	mydomain.com/Customer	controller = Customer, action = Index
2	mydomain.com/Customer>List	controller = Customer, action = List
3	mydomain.com/Customer>List>All	Нет соответствия: слишком много сегментов

Чем меньше сегментов мы получаем во входящем URL, тем больше мы полагаемся на значения по умолчанию, до того момента, пока мы не получим URL без сегментов, где используются только значения по умолчанию. Вы можете увидеть результат использования значений по умолчанию, если снова запустите приложение. На этот раз, когда браузер запрашивает корневой URL для приложения, будут использоваться значения по умолчанию для сегментных переменных controller и action, что приведет к тому, что MVC вызовет метод действия Index контроллера Home, как показано на рисунке 13-4.

**Рисунок 13-4:** Использование значений по умолчанию, чтобы расширить сферу использования роута



### Юнит тестирование: значения по умолчанию

Нам не нужно предпринимать никаких специальных действий, если мы используем вспомогательные методы для проверки роутов, которые определяют значения по умолчанию. Вот мы сделали изменения в тестовом методе TestIncomingRoutes в файле RouteTests.cs для роута, который мы определили в листинге 13-10:

```
...
[TestMethod]
public void TestIncomingRoutes() {
    TestRouteMatch("~/", "Home", "Index");
    TestRouteMatch("~/Customer", "Customer", "Index");
    TestRouteMatch("~/Customer/List", "Customer", "List");
    TestRouteFail("~/Customer/List/All");
}
...
```

Единственное замечание состоит в том, что мы должны указать URL по умолчанию как `~/`, так как таким образом ASP.NET представляет URL системе маршрутизации. Если указать пустую строку (""), которую мы использовали для определения роута, или `/`, система маршрутизации выбросит исключение, и тест будет провален.

# Использование статических URL сегментов

Не все сегменты в URL паттерне должны быть переменными. Вы также можете создавать паттерны, которые имеют статические сегменты. Предположим, нам нужно соответствие с URL вроде этого для поддержки URL, которые начинаются с `Public`:

```
http://mydomain.com/Public/Home/Index
```

Мы можем сделать это с помощью такого паттерна, как тот, что показан в листинге 13-11.

**Листинг 13-11:** URL паттерн со статическими сегментами

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
namespaceUrlsAndRoutes
{
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.MapRoute("MyRoute", "{controller}/{action}",
                new { controller = "Home", action = "Index" });
            routes.MapRoute("", "Public/{controller}/{action}",
                new { controller = "Home", action = "Index" });
        }
    }
}
```

Этот новый паттерн будет соответствовать только тем URL, которые содержат три сегмента, первый из которых обязательно должен быть `Public`. Два других сегмента могут содержать любые значения и будут использоваться для переменных `controller` и `action`. Если последние два сегмента опущены, то будут использоваться значения по умолчанию.

Мы также можем создавать URL паттерны, которые имеют сегменты, содержащие как статические элементы, так и элементы переменных, как показанный в листинге 13-12.

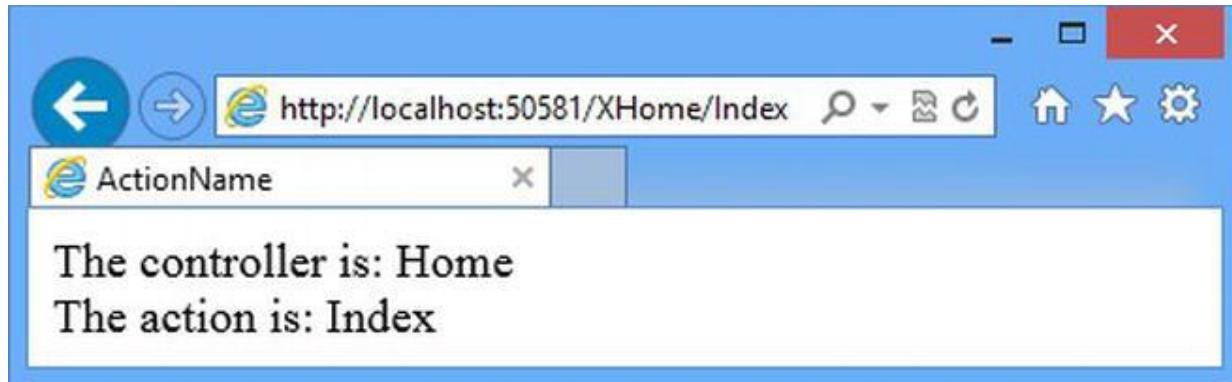
**Листинг 13-12:** URL паттерн со смешанным сегментом

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
namespaceUrlsAndRoutes
{
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.MapRoute("", "X{controller}/{action}");
            routes.MapRoute("MyRoute", "{controller}/{action}",
                new { controller = "Home", action = "Index" });
            routes.MapRoute("", "Public/{controller}/{action}",
                new { controller = "Home", action = "Index" });
        }
    }
}
```

```
    }
}
```

Паттерн в этом роуте подходит любому двухсегментному URL, где первый сегмент начинается на букву x. Значение для controller взято из первого сегмента, за исключением x. Значение action берется из второго сегмента. Вы можете увидеть результат использования этого роута, если запустите приложение и перейдите по /XHome/Index. Результат показан на рисунке 13-5.

**Рисунок 13-5:** Статические элементы и элементы переменных в одном сегменте



## Порядок роутов

В листинге 13-12 мы определили новый роут и разместили его перед всеми другими в методе RegisterRoutes. Мы сделали это, потому что роуты применяются в том порядке, в котором они появляются в объекте RouteCollection. Метод MapRoute добавляет роут в конец коллекции, что обозначает, что роуты обычно применяются в том порядке, в котором их добавляют. Мы говорим, "обычно", потому что есть методы, которые позволяют нам вставлять роуты в определенные места. Мы, как правило, не используем эти методы, потому что применение роутов в том порядке, в котором они определены, упрощает понимание маршрутизации для приложения. Система маршрутизации пытается сопоставить входящий URL с URL паттерном роута, который был определен первым, и переходит к следующему роуту, только если совпадения нет. И так система проходит по очереди по роутам, пока не будет найдено совпадение или пока не закончатся роуты. Поэтому сперва мы должны определять самые конкретные роуты. Роут, который мы добавили в листинге 13-12, является более конкретным, чем роут, который за ним следует. Предположим, что мы отменили порядок роутов, например:

```
...
routes.MapRoute("MyRoute", "{controller}/{action}",
new { controller = "Home", action = "Index" });
routes.MapRoute("", "X{controller}/{action}");
...

```

Тогда будет использоваться первый роут, который соответствует любому URL с нулем, одним или двумя сегментами. Более конкретный роут, который сейчас находится на втором месте в списке, никогда не будет достигнут. Новый маршрут исключает ведущую x в URL, но это не будет сделано более старым роутом, таким образом, URL, такие как этот:

```
http://mydomain.com/XHome/Index
```

будут нацелены на контроллер XHome, который не существует, и это приведет к ошибке 404—Not Found, которая будет отправлена пользователю.

Если вы еще не читали раздел о модульном тестировании входящих URL, мы предлагаем вам сделать это сейчас. Если вы проводите модульное тестирование только одной части вашего MVC приложения, то это должна быть URL схема.

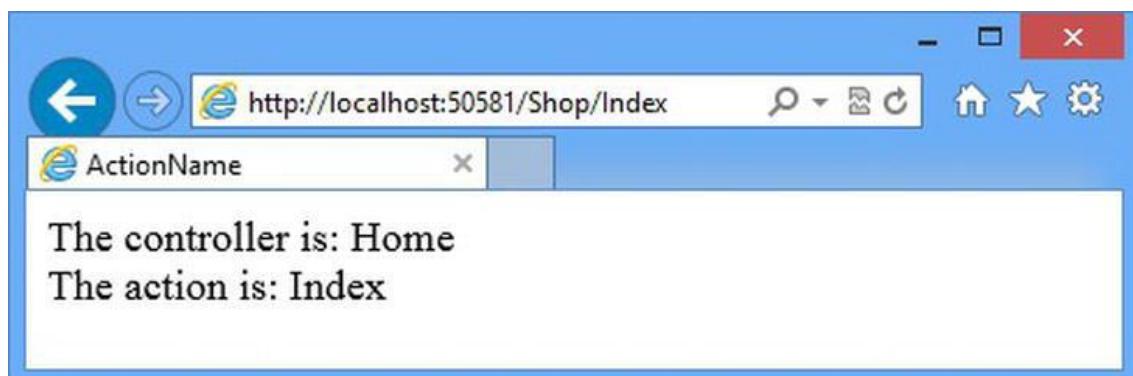
Мы можем объединять статические URL сегменты и значения по умолчанию для создания алиаса для конкретного URL. Это может быть полезно, если вы опубликовали URL схему публично, и с ней работает пользователь. Если вы в этой ситуации вы проводите рефакторинг приложения, вы должны сохранить прежний формат URL так, чтобы любые избранные URL или макро скрипты, которые создал пользователь, продолжали работать. Давайте представим, что у нас был контроллер `Shop`, который в настоящее время заменен контроллером `Home`. В листинге 13-13 показано, как мы можем создать роут, чтобы сохранить старую URL схему.

**Листинг 13-13:** Смешивание статических URL сегментов и значений по умолчанию

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
namespaceUrlsAndRoutes
{
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.MapRoute("ShopSchema", "Shop/{action}",
                new { controller = "Home" });
            routes.MapRoute("", "X{controller}/{action}");
            routes.MapRoute("MyRoute", "{controller}/{action}",
                new { controller = "Home", action = "Index" });
            routes.MapRoute("", "Public/{controller}/{action}",
                new { controller = "Home", action = "Index" });
        }
    }
}
```

Роут, который мы добавили, соответствует любому двухсегментному URL, где первым сегментом является `Shop`. Значение `action` берется из второго сегмента URL. URL паттерн не содержит сегмента для `controller`, поэтому используется данное нами значение по умолчанию. Это означает, что запрос для действий контроллера `Shop` переводится в запрос для контроллера `Home`. Вы можете увидеть результат использования этого роута, если запустите приложение и перейдете по URL `/Shop/Index`. Как показано на рисунке 13-6, роут, который мы добавили, заставляет MVC нацеливаться на действие `Index` в контроллере `Home`.

**Рисунок 13-6:** Создание алиаса для сохранения URL схемы



И мы можем пойти еще дальше и создать алиасы для методов действий, которые также были переделаны и больше не присутствует в контроллере. Чтобы сделать это, мы просто создаем статический URL и добавляем значения controller и action в качестве значений по умолчанию, как показано в листинге 13-14.

**Листинг 13-14:** Создание алиасов для контроллера и действия

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
namespaceUrlsAndRoutes
{
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.MapRoute("ShopSchema2", "Shop/OldAction",
                new { controller = "Home", action = "Index" });
            routes.MapRoute("ShopSchema", "Shop/{action}",
                new { controller = "Home" });
            routes.MapRoute("", "X{controller}/{action}");
            routes.MapRoute("MyRoute", "{controller}/{action}",
                new { controller = "Home", action = "Index" });
            routes.MapRoute("", "Public/{controller}/{action}",
                new { controller = "Home", action = "Index" });
        }
    }
}
```

Обратите внимание еще раз, что мы разместили наш новый роут так, чтобы он определялся в первую очередь. Это потому что он является более конкретным, чем роуты, которые следуют за ним. Если бы запрос для Shop/OldAction был обработан, например, следующим роутом, мы получили бы другой результат, а не тот, что мы хотим. Запрос бы обработался с ошибкой 404–Not Found, а не переведен, чтобы сохранить контакт с нашими клиентами.

### Юнит тест: тестирование статических сегментов

Еще раз, мы можем использовать вспомогательные методы для роутов, URL паттерны которых содержат статические сегменты. Вот что мы добавили в тестовый метод TestIncomingRoutes, чтобы протестировать роут, добавленный в листинге 13-14:

```
...
[TestMethod]
public void TestIncomingRoutes() {
    TestRouteMatch("~/", "Home", "Index");
    TestRouteMatch("~/Customer", "Customer", "Index");
    TestRouteMatch("~/Shop/Index", "Home", "Index");
    TestRouteMatch("~/Customer/List", "Customer", "List");
    TestRouteFail("~/Customer/List/All");
}
...
```

# Определение пользовательских переменных для сегментов

Сегментные переменные `controller` и `action` имеют особое значение для MVC и, очевидно, что они соответствуют контроллеру и методу действия, которые будут использоваться для обработки запроса. Мы не ограничены этими встроенными сегментными переменными: мы также можем определить наши собственные переменные, как показано в листинге 13-15. (Мы удалили все существующие роуты из предыдущего раздела, чтобы начать все сначала).

**Листинг 13-15:** Определение дополнительных переменных в URL паттерне

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
namespaceUrlsAndRoutes
{
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.MapRoute("MyRoute", "{controller}/{action}/{id}",
                new
                {
                    controller = "Home",
                    action = "Index",
                    id = "DefaultId"
                });
        }
    }
}
```

Роут URL паттерна определяет стандартные переменные `controller` и `action`, а также пользовательскую переменную `id`. Этот роут будет соответствовать любому URL с количеством сегментов от нуля до трех. Содержание третьего сегмента будет присвоено переменной `id`, а если третьего сегмента нет, будет использоваться значение по умолчанию.

## Внимание

Некоторые имена зарезервированы и не доступны для имен пользовательских переменных сегмента. Это `controller`, `action` и `area`. Смысл первых двух очевиден, и мы объясним роль областей в следующей главе.

Мы можем получить доступ к любой сегментной переменной в методе действий с помощью свойства `RouteData.Values`. Чтобы продемонстрировать это, мы добавили метод действия `CustomVariable` в класс `HomeController`, как показано в листинге 13-16.

**Листинг 13-16:** Доступ к пользовательской переменной сегмента в методе действия

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
namespaceUrlsAndRoutes.Controllers
```

```

{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewBag.Controller = "Home";
            ViewBag.Action = "Index";
            return View("ActionName");
        }
        public ActionResult CustomVariable()
        {
            ViewBag.Controller = "Home";
            ViewBag.Action = "CustomVariable";
            ViewBag.CustomVariable = RouteData.Values["id"];
            return View();
        }
    }
}

```

Этот метод получает значение пользовательской переменной в роутовом URL паттерне и передает его представлению при помощи ViewBag. Щелкните правой кнопкой мыши по новому методу действия в редакторе кода и выберите Add View. Назовите представление CustomVariable и нажмите кнопку Add. Visual Studio создаст новый файл CustomVariable.cshtml в папке /Views/Home. Измените представление так, чтобы оно соответствовало содержанию, показанному в листинге 13-17.

### Листинг 13-17: Отображение значения пользовательской сегментной переменной

```

@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Custom Variable</title>
</head>
<body>
    <div>The controller is: @ViewBag.Controller</div>
    <div>The action is: @ViewBag.Action</div>
    <div>The custom variable is: @ViewBag.CustomVariable</div>
</body>
</html>

```

Чтобы увидеть результат использования пользовательской переменной сегмента, запустите приложение и перейдите по URL /Home/CustomVariable>Hello. Вызывается метод действия CustomVariable в контроллере Home, и значение пользовательской переменной сегмента извлекается из ViewBag и передается представлению. Вы можете увидеть результат на рисунке 13-7.

### Рисунок 13-7: Отображение значения пользовательской сегментной переменной



Мы предоставили значение по умолчанию для переменной сегмента `id`, что означает, что вы увидите результаты, показанные на рисунке 13-8, если вы перейдете на `/Home/CustomVariable`.

**Рисунок 13-8:** Значение по умолчанию для пользовательской сегментной переменной



### Юнит тест: тестирование пользовательских сегментных переменных

Мы включили поддержку для тестирования пользовательских сегментных переменных сегмента во вспомогательные методы тестирования. Метод `TestRouteMatch` имеет дополнительный параметр, который принимает анонимный тип, содержащий имена свойств, которые мы хотим протестировать, и ожидаемые значения. Вот изменения, которые мы сделали для тестового метода `TestIncomingRoutes`, чтобы проверить роут, определенный в листинге 13-15:

```
...
[TestMethod]
public void TestIncomingRoutes() {
    TestRouteMatch("~/", "Home", "Index", new { id = "DefaultId" });
    TestRouteMatch("~/Customer", "Customer", "index", new { id = "DefaultId" });
    TestRouteMatch("~/Customer/List", "Customer", "List",
        new { id = "DefaultId" });
    TestRouteMatch("~/Customer/List/All", "Customer", "List", new { id = "All" });
    TestRouteFail("~/Customer/List/All/Delete");
}
...
```

### Использование пользовательских переменных в качестве параметра метода действия

Использование свойства `RouteData.Values` является только одним способом доступа к пользовательским переменным роута. Другой способ гораздо более элегантный. Если мы определим параметры нашего метода действия именами, которых соответствуют переменным URL паттерна, MVC передаст значения, полученные из URL, в качестве параметров методу действия. Например, пользовательская переменная, которая была определена в роуте в листинге 13-15, называется `id`. Мы можем изменить метод действия `CustomVariable`, так чтобы он имел соответствующий параметр, как показано в листинге 13-18.

**Листинг 13-8:** Изменение метода действия так, чтобы сегментная переменная соответствовала его параметру

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
```

```

namespaceUrlsAndRoutes.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewBag.Controller = "Home";
            ViewBag.Action = "Index";
            return View("ActionName");
        }
        public ActionResult CustomVariable(string id)
        {
            ViewBag.Controller = "Home";
            ViewBag.Action = "CustomVariable";
            ViewBag.CustomVariable = id;
            return View();
        }
    }
}

```

Когда система маршрутизации находит соответствие URL с определенным нами в листинге 13-15 роутом, значение третьего сегмента в URL присваивается пользовательской переменной `id`. MVC сравнивает список сегментных переменных со списком параметров метода действия, и если имена совпадают, передает значения из URL в метод.

Мы определили параметр `id` как `string`, но MVC попытается преобразовать URL значение в тот тип параметра, который мы определим. Если мы объявили параметр `id` как `int` или `DateTime`, то мы получим из URL значение, разобранное как экземпляр этого типа. Это элегантная и полезная возможность, которая устраняет необходимость самим проводить преобразование типов.

#### **Примечание**

*MVC использует систему модели связывания данных для преобразования значений, содержащихся в URL, в .NET типы и может обрабатывать гораздо более сложные ситуации, чем показанная в этом примере. Мы расскажем о модели связывания данных в главе 22.*

---

## **Определение дополнительных URL сегментов**

Дополнительный сегмент URL это тот, который пользователь не должен указывать, но для которого не задано значение по умолчанию. В листинге 13-19 показан пример, и мы указываем, что переменная сегмента не является обязательной, установив значение по умолчанию `UrlParameter.Optional`.

#### **Листинг 13-19:** Указание дополнительного URL сегмента

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
namespaceUrlsAndRoutes
{
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {

```

```
        routes.MapRoute("MyRoute", "{controller}/{action}/{id}",  
new  
{  
    controller = "Home",  
    action = "Index",  
    id = UrlParameter.Optional  
});  
}  
}  
}
```

Этот роут будет соответствовать URL, независимо от того, был ли указан `id` сегмент. В таблице 13-3 показано, как это работает с различными URL.

**Таблица 13-3:** Соответствие URL с дополнительной сегментной переменной

Число сегментов	Пример	Соответствие
0	mydomain.com	controller = Home, action = Index
1	mydomain.com/Customer	controller = Customer, action = Index
2	mydomain.com/Customer>List	controller = Customer, action = List
3	mydomain.com/Customer>List>All	controller = Customer, action = List, id = All
4	mydomain.com/Customer>List>All>Delete	Нет соответствия: СЛИШКОМ МНОГО сегментов

Как вы можете видеть из таблицы, переменная `id` добавляется в набор переменных только при наличии соответствующего сегмента во входящем URL. Эта функция полезна, если вам нужно знать, задал ли пользователь значение для сегмента переменной. Если значение не было предоставлено для дополнительной сегментной переменной, значение соответствующего параметра будет `null`. В листинге 13-20 мы обновили наш контроллер, чтобы он реагировал, если для сегментной переменной `id` не было предоставлено значение.

### **Листинг 13-20:** Проверка на то, было ли предоставлено значение для дополнительной сегментной переменной

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
namespaceUrlsAndRoutes.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewBag.Controller = "Home";
            ViewBag.Action = "Index";
            return View("ActionName");
        }
        public ActionResult CustomVariable(string id)
        {
            ViewBag.Controller = "Home";
            ViewBag.Action = "CustomVariable";
            ViewBag.CustomVariable = id == null ? "<no value>" : id;
            return View();
        }
    }
}
```

```
    }  
}
```

Вы можете увидеть результат, если запустите приложение и перейдете по /Home/CustomVariable (здесь не определено значение сегментной переменной `id`), на рисунке 13-9.

**Рисунок 13-9:** Определение того, что URL не содержит значение для дополнительной сегментной переменной



### Использование дополнительных URL сегментов для вынужденного разделения понятий

Некоторые разработчики, которые очень сосредоточены на разделении понятий в MVC паттерне, не любят устанавливать значения по умолчанию для переменных сегмента в роутах для приложения. Если это касается и вас, вы можете использовать дополнительные параметры C# наряду с дополнительной переменной сегмента в роуте, чтобы определить значения по умолчанию для параметров метода действия. В качестве примера в листинге 13-21 показано, как мы изменили метод действия `CustomVariable`, чтобы определить значение по умолчанию для параметра `id`, который будет использоваться, если URL не содержит значения.

**Листинг 13-21:** Определение значения по умолчанию для параметра метода действия

```
...  
public ActionResult CustomVariable(string id = "DefaultId") {  
    ViewBag.Controller = "Home";  
    ViewBag.Action = "CustomVariable";  
    ViewBag.CustomVariable = id;  
    return View();  
}  
...
```

Для параметра `id` всегда будет значение (либо из URL, либо по умолчанию), поэтому мы удалили код, который имеет дело со значением `null`. Этот метод действия совместно с роутом, который мы определили в листинге 13-19, функционально эквивалентен роуту, который мы определили в листинге 13-15:

```
...  
routes.MapRoute("MyRoute", "{controller}/{action}/{id}",  
    new { controller = "Home", action = "Index", id = "DefaultId" });  
...
```

Разница заключается в том, что значение по умолчанию для сегментной переменной `id` обозначается в коде контроллера, а не в определении роута.

### Юнит тестирование: дополнительные URL сегменты

Единственный вопрос, который необходимо учитывать при тестировании дополнительных URL сегментов, заключается в том, что сегментная переменная не будет добавлена в коллекцию `RouteData.Values`, если значение не было найдено в URL. Это означает, что вам не следует включать переменную в анонимный тип, если вы тестируете URL, который содержит дополнительный сегмент. Вот наши изменения тестового метода `TestIncomingRoutes` для роута, определенного в листинге 13-19.

```
...
[TestMethod]
public void TestIncomingRoutes() {
    TestRouteMatch("~/", "Home", "Index");
    TestRouteMatch("~/Customer", "Customer", "index");
    TestRouteMatch("~/Customer/List", "Customer", "List");
    TestRouteMatch("~/Customer/List/All", "Customer", "List", new { id = "All" });
    TestRouteFail("~/Customer/List/All/Delete");
}
...
...
```

## Определение роутов с разным числом сегментов

Другой способ изменить стандартный консерватизм URL паттернов – это принять число переменных URL сегментов. Это позволяет принимать URL произвольной длины в одном роуте. Вы определяете поддержку переменных сегмента, обозначив одну из сегментных переменных как `catchall`, что делается при помощи добавления звездочки (\*), как показано в листинге 13-22.

### Листинг 13-22: Назначение `catchall` переменной

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
namespaceUrlsAndRoutes
{
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{*catchall}",
                new
                {
                    controller = "Home",
                    action = "Index",
                    id = UrlParameter.Optional
                });
        }
    }
}
```

Мы расширили роут из предыдущего примера, чтобы добавить переменную сегмента `catchall`, которую мы образно назвали `catchall`. Этот роут теперь будет соответствовать любому URL, независимо от количества сегментов, которые он содержит, или значения любого из этих сегментов. Первые три сегмента используются для установки значений для переменных `controller`, `action` и `id` соответственно. Если URL содержит дополнительные сегменты, все они назначаются переменной `catchall`, как показано в таблице 13-4.

**Таблица 13-4:** Соответствие URL с сегментной переменной *catchall*

Число сегментов	Пример	Соответствие
0	/	controller = Home, action = Index
1	/Customer	controller = Customer, action = Index
2	/Customer>List	controller = Customer, action = List
3	/Customer>List>All	controller = Customer, action = List, id = All
4	/Customer>List>All>Delete	controller = Customer, action = List, id = All, catchall = Delete
5	/Customer>List>All>Delete/Perm	controller = Customer, action = List, id = All, catchall = Delete/Perm

Здесь нет верхнего ограничения по числу сегментов, поэтому URL паттерн данного роута будет совпадать с любым подходящим URL. Обратите внимание, что сегменты, находящиеся в *catchall*, представлены в форме *segment/segment/segment*. Мы должны обрабатывать строку, чтобы получить отдельные сегменты.

### Юнит тест: тестирование сегментных переменных *catchall*

Мы можем обрабатывать переменную *catchall* как пользовательскую переменную. Разница лишь в том, что мы должны ожидать несколько сегментов, которые связываются в одно значение, например, *segment/segment/segment*. Обратите внимание, что мы не получим начальный или конечный знаки /. Вот изменения метода *TestIncomingRoutes*, которые демонстрируют тестирование сегмента *catchall*. Тут используется роут, определенный в листинге 13-22, и URL, показанные в таблице 13-4:

```
...
[TestMethod]
public void TestIncomingRoutes() {
    TestRouteMatch("~/", "Home", "Index");
    TestRouteMatch("~/Customer", "Customer", "Index");
    TestRouteMatch("~/Customer/List", "Customer", "List");
    TestRouteMatch("~/Customer/List>All", "Customer", "List", new { id = "All" });
    TestRouteMatch("~/Customer/List>All>Delete", "Customer", "List",
        new { id = "All", catchall = "Delete" });
    TestRouteMatch("~/Customer/List>All>Delete/Perm", "Customer", "List",
        new { id = "All", catchall = "Delete/Perm" });
}
...
```

### Определение приоритета контроллера по пространству имен

Если входящий URL соответствует роуту, MVC фреймворк берет значение переменной *controller* и ищет соответствующее имя. Например, если значение переменной *controller* равно *Home*, то MVC ищет контроллер *HomeController*. Это *неполное* имя класса, то есть MVC не знает, что делать, если есть два или более класса *HomeController* в разных пространствах имен.

Чтобы понять эту проблему, создайте новую папку в корневом каталоге проекта, назовите ее *AdditionalControllers* и добавьте новый контроллер *Home*. Установите содержание, чтобы оно соответствовало листингу 13-23.

### Листинг 13-23: Добавление второго контроллера *Home* в проект

```
using System;
using System.Collections.Generic;
```

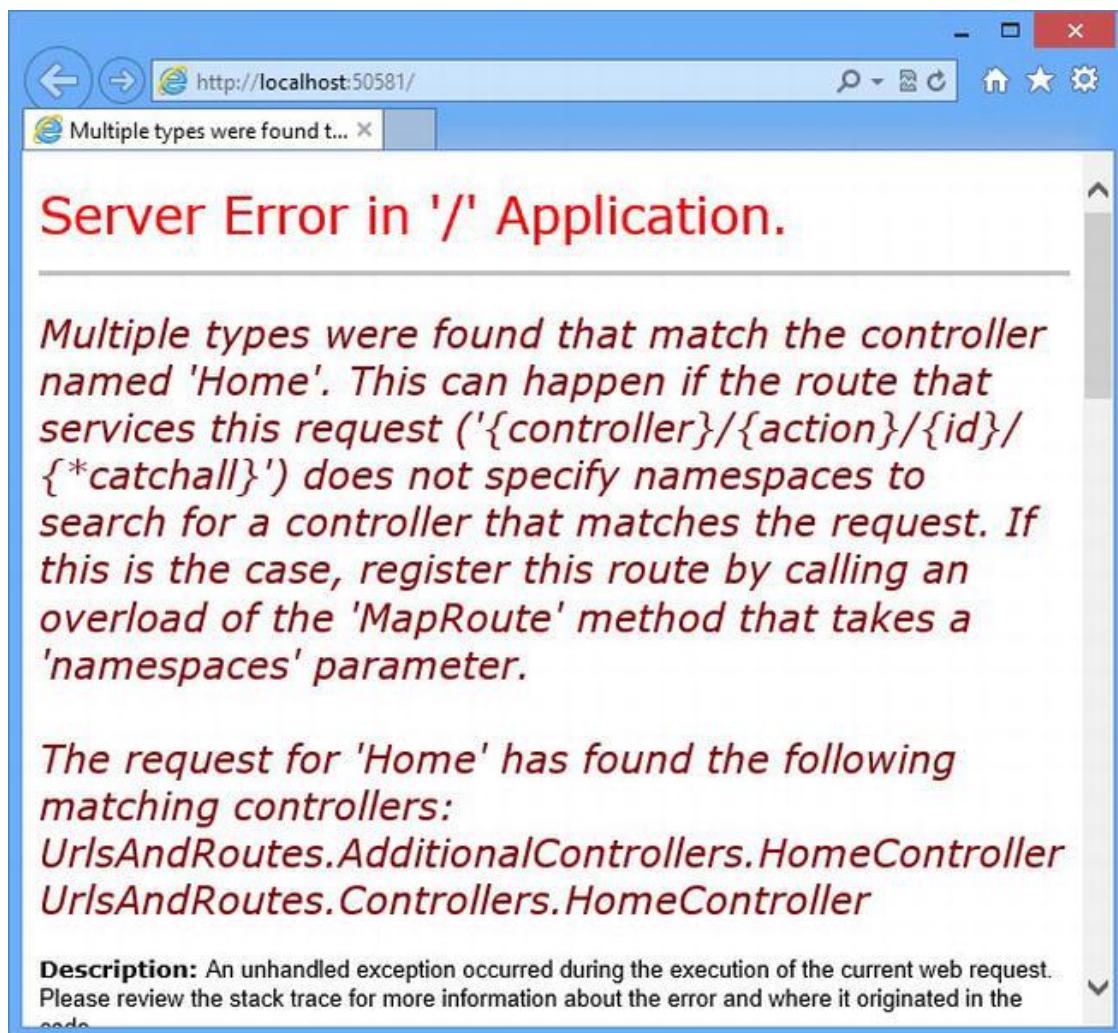
```

using System.Linq;
using System.Web;
using System.Web.Mvc;
namespaceUrlsAndRoutes.AdditionalControllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewBag.Controller = "Additional Controllers - Home";
            ViewBag.Action = "Index";
            return View("ActionName");
        }
    }
}

```

Если вы запустите приложение, вы увидите ошибку, показанную на рисунке 13-10.

**Рисунок 13-10:** Возникает ошибка, если есть два контроллера с одним и тем же именем



MVC фреймворк искал класс `HomeController` и нашел два: один в нашем исходном проекте и один в нашем проекте `AdditionalControllers`. Если вы прочтете текст ошибки, показанной на рисунке 13-10, вы увидите, что MVC фреймворк услужливо сообщает нам, какие классы он нашел.

Эта проблема возникает чаще, чем вы можете себе представить, особенно если вы работаете над большим MVC проектом, который использует библиотеки контроллеров других команд

разработчиков или сторонних программистов. Например, вполне естественно назвать контроллер, связанный с учетными записями пользователей, `AccountController`, и это только вопрос времени, когда вы столкнетесь с конфликтом имен.

Чтобы решить эту проблему, мы можем сказать MVC фреймворку отдавать предпочтение определенным пространствам имен при попытке выбрать имя класса контроллера, как показано в листинге 13-24.

**Листинг 13-24:** Указание порядка приоритета пространств имен

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
namespace URLsAndRoutes
{
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{*catchall}",
                new
                {
                    controller = "Home",
                    action = "Index",
                    id = UrlParameter.Optional
                },
                new[] { "URLsAndRoutes.AdditionalControllers" });
        }
    }
}
```

Мы выражаем пространства имен как массив строк, и в листинге мы сказали MVC смотреть сначала на пространство имен `URLsAndRoutes.AdditionalControllers`, прежде чем искать в другом месте.

Если подходящий контроллер не может быть найден в этом пространстве имен, тогда MVC по вернется к своему поведению по умолчанию и будет искать совпадение в доступных пространствах имен. Если вы запустите приложение после этого дополнения, вы увидите результат, продемонстрированный на рисунке 13-11. Здесь показан запрос для корневого URL, который переводится в запрос для метода действия `Index` контроллера `Home`. Это запрос был отправлен контроллеру, который мы определили в пространстве имен `AdditionalControllers`.

**Рисунок 13-11:** Приоритет контроллера в определенном пространстве имен



Пространства имен, добавленные в роут, обладают равным приоритетом. MVC не проверяет первое пространство имен, прежде чем перейти ко второму и так далее. Например, предположим, что мы добавили оба наших пространства имен в роут:

```
...
routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{*catchall}",
    new { controller = "Home", action = "Index",
        id = UrlParameter.Optional
    },
    new[] { "URLsAndRoutes.AdditionalControllers", "UrlsAndRoutes.Controllers" });
...
...
```

Мы увидели бы ту же ошибку, что показана на рисунке 13-10, потому что MVC пытается найти имя класса контроллера во всех пространствах имен, которые мы добавили к роуту. Если мы хотим отдать предпочтение одному контроллеру в одном пространстве имен, но и не хотим иметь проблемы с контроллерами в других пространствах имен, нам нужно создать несколько роутов, как показано в листинге 13-25.

**Листинг 13-25:** Использование нескольких роутов для выбора нужного пространства имен

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
namespace URLsAndRoutes
{
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.MapRoute("AddControllerRoute", "Home/{action}/{id}/{*catchall}",
                new
                {
                    controller = "Home",
                    action = "Index",
                    id = UrlParameter.Optional
                },
                new[] { "URLsAndRoutes.AdditionalControllers" });
            routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{*catchall}",
                new
                {
                    controller = "Home",
                    action = "Index",
                    id = UrlParameter.Optional
                },
                new[] { "URLsAndRoutes.Controllers" });
        }
    }
}
```

Наш первый роут применяется тогда, когда пользователь явно запрашивает URL, первым сегментом которого является `Home`, и будет нацелен на контроллер `Home` в папке `AdditionalControllers`. Все другие запросы, в том числе те, где не указан первый сегмент, будут обработаны контроллерами в папке `Controllers`.

Мы можем сказать MVC фреймворку искать только в пространствах имен, которые мы указали. Если соответствующий контроллер не может быть найден, то фреймворк не будет искать в другом месте. Листинг 13-26 показывает, как используется эта функция.

### Листинг 13-26: Отключение не используемых пространств имен

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
namespace URLsAndRoutes
{
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            Route myRoute = routes.MapRoute("AddControllerRoute",
                "Home/{action}/{id}/{*catchall}",
                new
                {
                    controller = "Home",
                    action = "Index",
                    id = UrlParameter.Optional
                },
                new[] { "URLsAndRoutes.AdditionalControllers" });
            myRoute.DataTokens["UseNamespaceFallback"] = false;
        }
    }
}
```

Метод `MapRoute` возвращает объект `Route`. Мы игнорировали это в предыдущих примерах, потому что нам не нужно было вносить любые изменения в роуты, которые были созданы. Чтобы отключить поиск контроллеров в других пространствах имен, мы берем объект `Route` и устанавливаем ключ `UseNamespaceFallback` в свойстве коллекции `DataTokens` на `false`.

Эта настройка будет передана компоненту, отвечающему за поиск контроллеров, который известен как фабрика контроллеров и который мы подробно обсудим в главе 17. Результат этого дополнения заключается в том, что запросы, которые не могут быть удовлетворены контроллером `Home` в папке `AdditionalControllers`, не сработают.

## Ограничение роутов

В начале главы мы рассказали, что URL паттерны консервативны в том, что касается соответствию с сегментами, и либеральны в том, что касается соответствия содержанию сегментов. В нескольких предыдущих разделах мы объяснили различные способы управления степенью консерватизма: создание роутов, которые соответствуют большему или меньшему числу сегментов, при помощи значений по умолчанию, дополнительных переменных и так далее.

Сейчас настало время рассмотреть, как мы можем контролировать либерализм, который заключается в соответствие содержанию URL сегментов: как ограничить набор URL, которым будет соответствовать роут. Как только мы будем контролировать оба эти аспекта поведения роута, мы сможем создавать URL схемы, которые выражены с идеальной точностью.

### Ограничение роута при помощи регулярного выражения

Первый способ, который мы рассмотрим, заключается в ограничении роута при помощи регулярных выражений. Листинг 13-27 содержит пример.

### Листинг 13-27: Использование регулярного выражения для ограничения роута

```
...
public static void RegisterRoutes(RouteCollection routes) {
    routes.MapRoute("MyRoute", "{controller}/{action}/{id}/*catchall",
        new { controller = "Home", action = "Index", id = UrlParameter.Optional },
        new { controller = "^H.*" },
        new[] { "URLsAndRoutes.Controllers" });
}
...
```

Мы определяем ограничения путем передачи их в качестве параметра методу `MapRoute`. Как и значения по умолчанию, ограничения выражаются в виде анонимного типа, где свойства типа соответствуют именам сегментных переменных, которые мы хотим ограничить.

В этом примере мы использовали ограничение с помощью регулярных выражений. Оно заключается в том, что URL и роут соответствуют друг другу только тогда, когда значение переменной `controller` начинается с буквы `H`.

#### Примечание

**Значения по умолчанию используются до проверки на ограничения. Так, например, если мы запрашиваем URL /, для controller применяется значение по умолчанию Home. Затем проверяются ограничения, и поскольку значение controller начинается с H, то URL по умолчанию будет соответствовать роуту.**

## Ограничение роута до набора указанных значений

Мы можем использовать регулярные выражения, чтобы ограничить роут таким образом, что только конкретные значения URL сегмента будут способствовать соответствуию URL и роута. Это можно сделать, используя символ вертикальной черты (`|`), как показано в листинге 13-28.

### Листинг 13-28: Ограничение роута до определенного набора значений сегментных переменных

```
...
public static void RegisterRoutes(RouteCollection routes) {
    routes.MapRoute("MyRoute", "{controller}/{action}/{id}/*catchall",
        new { controller = "Home", action = "Index", id = UrlParameter.Optional },
        new { controller = "^H.*", action = "^Index$|^About$" },
        new[] { "URLsAndRoutes.Controllers" });
}
...
```

Это ограничение позволяет роуту соответствовать только тем URL, где значениями сегмента действия являются `Index` или `About`. Ограничения применяются вместе, так что ограничения, накладываемые на значение переменной `action`, сочетаются со значениями, накладываемыми на переменную `controller`. Это означает, что роут в листинге 13-28 будет только в том случае соответствовать URL, если переменная `controller` начинается с буквы `H`, а для переменной `action` есть значения `Index` или `About`. Итак, теперь вы видите, что мы понимаем под созданием идеально точных роутов.

## Ограничение роутов при помощи HTTP методов

Мы можем ограничить роуты, чтобы они соответствовали URL только тогда, когда он запрашивается с помощью конкретного HTTP метода, как показано в листинге 13-29.

### Листинг 13-29: Ограничение роутов при помощи HTTP методов

```
...
public static void RegisterRoutes(RouteCollection routes) {
    routes.MapRoute("MyRoute", "{controller}/{action}/{id}/*catchall",
        new { controller = "Home", action = "Index", id = UrlParameter.Optional },
        new { controller = "^H.*", action = "Index|About",
            httpMethod = new HttpMethodConstraint("GET") },
        new[] { "URLsAndRoutes.Controllers" });
}
...
```

Формат для указания ограничения при помощи HTTP метода немного странный. Не имеет никакого значения, какое имя мы дали свойству, поскольку мы присвоили его экземпляру класса `HttpMethodConstraint`. В листинге мы назвали наше свойство для ограничения `HttpMethod`, чтобы его можно было отличить от других видов ограничений.

#### Примечание

**Возможность ограничения роутов при помощи HTTP методов не связана с возможностью ограничения методов действия при помощи таких атрибутов, как `HttpGet` и `HttpPost`. Ограничения роутов обрабатываются гораздо раньше в конвейере запросов, и они определяют имена контроллера и действия, необходимых для обработки запроса. Атрибуты методов действия используются для определения, какие конкретные методы действий будут использоваться для обработки запроса контроллером. Мы подробно расскажем о том, как работать с различными видами HTTP методов (в том числе таким, как `PUT` и `DELETE`), в главе 14.**

Мы передаем имена нужных HTTP методов в виде строковых параметров в конструктор класса `HttpMethodConstraint`. В листинге мы ограничили роут запросом GET, но мы могли бы легко добавить поддержку других методов, например:

```
...
httpMethod = new HttpMethodConstraint("GET", "POST") ,
...
```

### Юнит тестирование: ограничение роутов

При тестировании ограничения роутов важно проверить как URL, которые будут соответствовать роутам, так и URL, которые вы пытаетесь исключить, что можно сделать с помощью вспомогательных методов, представленных в начале главы. Вот изменения тестового метода `TestIncomingRoutes`, который мы использовали для проверки роута, определенного в листинге 13-29:

```
...
[TestMethod]
public void TestIncomingRoutes() {
    TestRouteMatch("~/", "Home", "Index");
    TestRouteMatch("~/Home", "Home", "Index");
    TestRouteMatch("~/Home/Index", "Home", "Index");
    TestRouteMatch("~/Home/About", "Home", "About");
    TestRouteMatch("~/Home/About/MyId", "Home", "About", new { id = "MyId" });
    TestRouteMatch("~/Home/About/MyId/More/Segments", "Home", "About",
        new {
            id = "MyId",
            catchall = "More/Segments"
    }
}
```

```

    });
    TestRouteFail("~/Home/OtherAction");
    TestRouteFail("~/Account/Index");
    TestRouteFail("~/Account/About");
}
...

```

## Определение пользовательского ограничения

Если стандартные ограничения не удовлетворяют ваши потребности, вы можете определить свои собственные ограничения путем реализации интерфейса `IRouteConstraint`. Чтобы продемонстрировать эту возможность, мы добавили в проект папку `Infrastructure` и создали новый файл класса `UserAgentConstraint.cs`, содержание которого показано в листинге 13-30.

### Листинг 13-30: Создание пользовательского ограничения роутов

```

using System.Web;
using System.Web.Routing;
namespaceUrlsAndRoutes.Infrastructure
{
    public class UserAgentConstraint : IRouteConstraint
    {
        private string requiredUserAgent;
        public UserAgentConstraint(string agentParam)
        {
            requiredUserAgent = agentParam;
        }
        public bool Match(HttpContextBase httpContext, Route route, string parameterName,
                           RouteValueDictionary values, RouteDirection routeDirection)
        {
            return httpContext.Request.UserAgent != null &&
                   httpContext.Request.UserAgent.Contains(requiredUserAgent);
        }
    }
}

```

Интерфейс `IRouteConstraint` определяет метод `Match`, который можно использовать для того, чтобы определить, было ли сделано нужное ограничение. Параметры метода `Match` обеспечивает доступ к запросу от клиента, роуту, который в настоящее время оценивается, имени параметра ограничения, сегментным переменным, извлеченным из URL, а также информации о том, является ли это запросом на проверку входящего или исходящего URL. В нашем примере мы проверяем значение свойства `UserAgent` запроса клиента, чтобы увидеть, если оно содержит значение, которое было передано в наш конструктор. Листинг 13-31 показывает пользовательское ограничение, которое используется для роута.

### Листинг 13-31: Применение пользовательского ограничения к роуту

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
usingUrlsAndRoutes.Infrastructure;
namespaceUrlsAndRoutes
{
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {

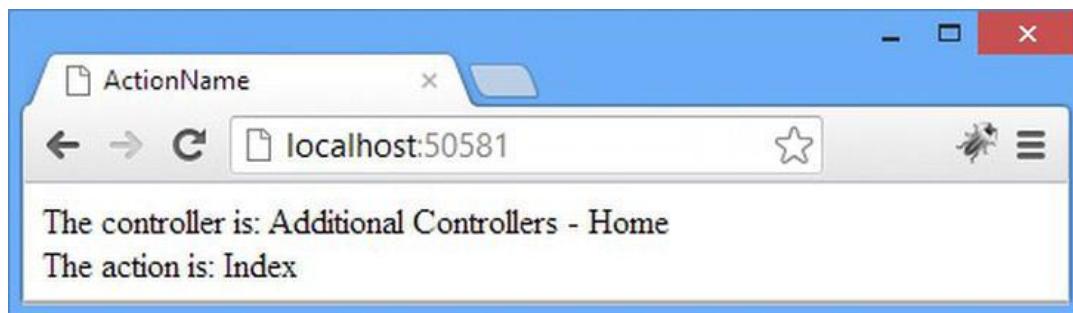
```

```
routes.MapRoute("ChromeRoute", "{*catchall}",
new { controller = "Home", action = "Index" },
new
{
    customConstraint = new UserAgentConstraint("Chrome")
},
new[] { "UrlsAndRoutes.AdditionalControllers" });
routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{*catchall}",
new
{
    controller = "Home",
    action = "Index",
    id = UrlParameter.Optional
},
new[] { "URLsAndRoutes.Controllers" });
}
```

В листинге мы ограничили первый роут так, что он будет соответствовать только запросы от браузеров, где строка пользовательского агента содержит Chrome. Если роут совпадет, то запрос будет отправлен методу действия `Index` в контроллере `Home`, определенном в папке `AdditionalControllers`, независимо от структуры и содержания URL, который был запрошен. Наш URL паттерн состоит только из переменной сегмента `catchall`, что означает, что значения переменных сегмента `controller` и `action` всегда будут браться из значений по умолчанию, а не из самого URL.

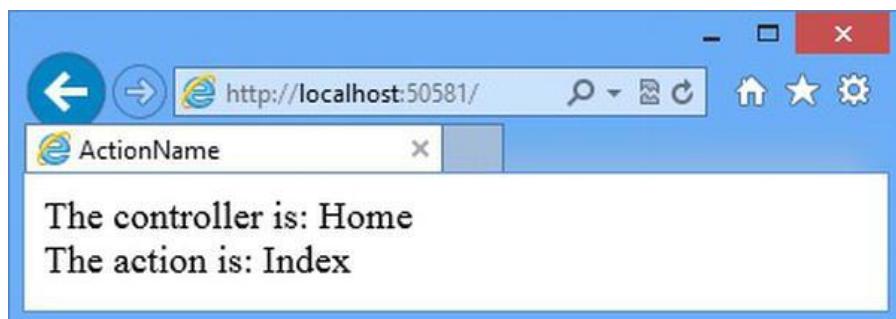
Второй роут будет соответствовать всем другим запросам и целевыми контроллерами в папке `Controllers`. Результат использования этих роутов заключается в том, что определенный вид браузера всегда останавливается на том же месте в приложении. Вы можете увидеть это на рисунке 13-12, где показан переход к приложению с помощью Google Chrome.

**Рисунок 13-12:** Переход к приложению при помощи Google Chrome



На рисунке 13-13 показан переход к приложению при помощи Internet Explorer.

**Рисунок 13-13:** Переход к приложению при помощи Internet Explorer



#### *Примечание*

*Внесем ясность, потому что иногда из-за этого мы получаем письма с жалобами: мы не предлагаем вам ограничивать приложение таким образом, чтобы оно поддерживало только один браузер. Мы использовали строки пользовательского агента только для того, чтобы продемонстрировать пользовательские ограничения роутов, и мы верим в равные возможности для всех браузеров. Мы действительно ненавидим веб сайты, которые пытаются навязать пользователям свои любимые браузеры.*

## Роутовые запросы для дисковых файлов

Не все запросы для MVC приложения предназначены для контроллеров и действий. Нам по-прежнему нужен способ обрабатывать такой контент, как изображения, статические HTML файлы, JavaScript библиотеки и так далее. В качестве демонстрации мы создали файл StaticContent.html в папке Content нашего MVC приложения, используя HTML Page. Листинг 13-32 показывает содержимое этого файла.

**Листинг 13-32:** Файл StaticContent.html

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head><title>Static HTML Content</title></head>
<body>
    This is the static html file (~/Content/StaticContent.html)
</body>
</html>
```

Система маршрутизации обеспечивает интегрированную поддержку для обслуживания такого содержания. Если вы запустите приложение и запросите URL /Content/StaticContent.html, вы увидите содержимое этого простого HTML файла, отображенного в браузере, как показано на рисунке 13-14.

**Рисунок 13-14:** Запрос файла со статическим содержанием



По умолчанию система маршрутизации проверяет URL на соответствие дисковому файлу перед оценкой роутов приложения, и именно поэтому нам не нужно было добавлять роут, чтобы получить результат, показанный на рисунке 13-14.

Если есть соответствие между запрашиваемым URL и дисковым файлом, то будет обработан дисковый файл, а роуты, определенные приложением, никогда не будут использованы. Мы можем изменить это поведение так, чтобы наши роуты оценивались раньше дисковых файлов, если мы установим свойство `RouteExistingFiles` `RouteCollection` на `true`, как показано в листинге 13-33.

#### Листинг 13-33: Включение оценки роутов до проверки файлов

```
public static void RegisterRoutes(RouteCollection routes) {  
    routes.RouteExistingFiles = true;  
    routes.MapRoute("ChromeRoute", "{*catchall}",  
        new { controller = "Home", action = "Index" },  
        new {  
            customConstraint = new UserAgentConstraint("Chrome")  
        },  
        new[] { "UrlsAndRoutes.AdditionalControllers" } );  
    routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{*catchall}",  
        new { controller = "Home", action = "Index",  
            id = UrlParameter.Optional },  
        new[] { "URLsAndRoutes.Controllers" } );  
}
```

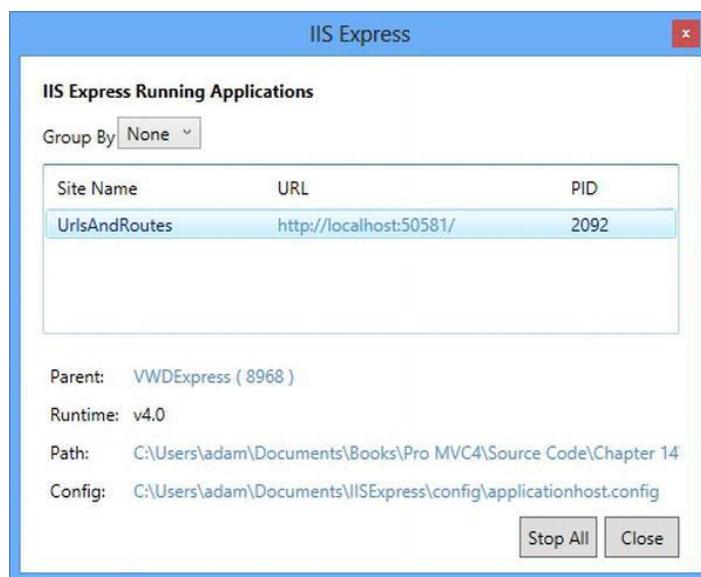
По соглашению это выражение размещается в верхней части метода `RegisterRoutes`, хотя все будет работать, даже если вы разместите его после определения роутов.

## Конфигурация сервера приложения

Visual Studio 2012 использует IIS Express в качестве сервера приложений для MVC проектов. Мы не только должны установить свойство `RouteExistingFiles` в методе `RegisterRoutes` на `true`, мы также должны сказать IIS Express не перехватывать запросы для дисковых, прежде чем они попадут в систему маршрутизации MVC.

Прежде всего, запустите IIS Express. Самый простой способ сделать это – это запустить MVC приложение из Visual Studio, которое покажет иконку IIS Express на панели задач. Щелкните правой кнопкой мыши по иконке и выберите из всплывающего меню `Show All Applications`. Нажмите на `UrlsAndRoutes` в столбце `Site Name` для отображения информации о конфигурации IIS, как показано на рисунке 13-15.

**Рисунок 13-15:** Информация о конфигурации IIS Express

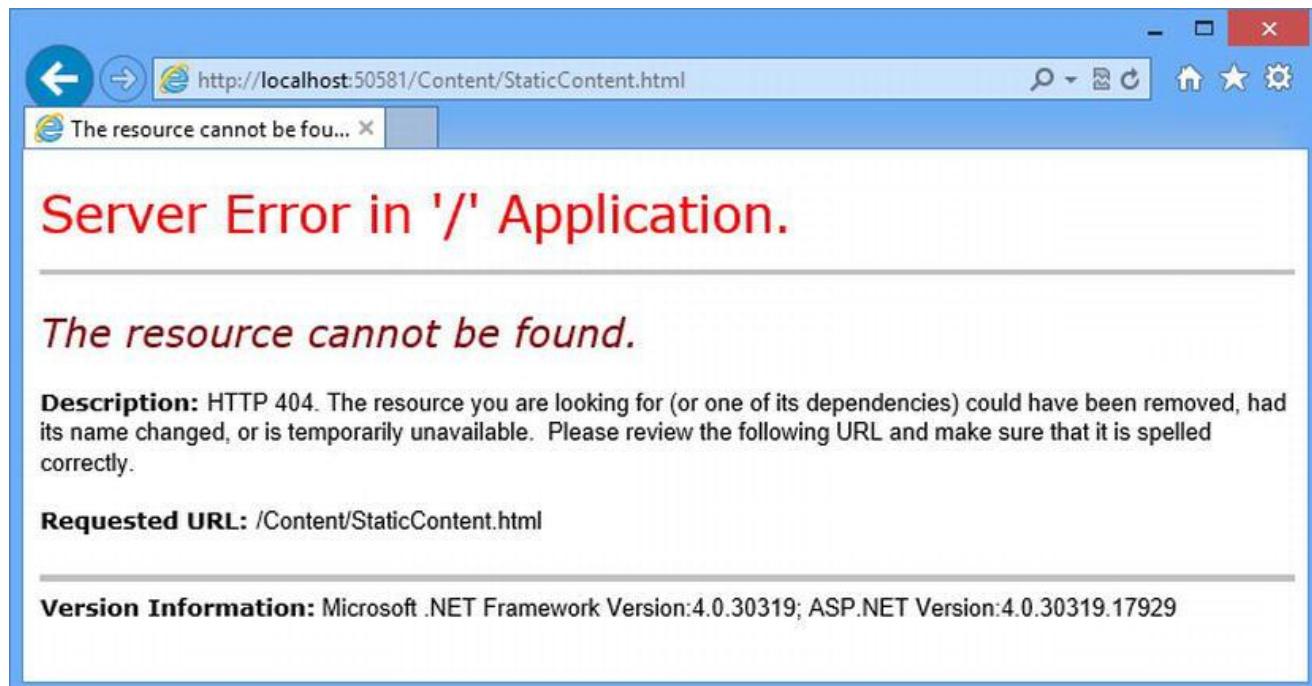


Нажмите на ссылку `Config` в нижней части окна, чтобы открыть конфигурационный файл IIS Express в Visual Studio. Теперь нажмите `Ctrl + F` и найдите `UrlRoutingModule-4.0`. Там будет запись, найденная в разделе `modules` конфигурационного файла, и нам нужно установить для атрибута `preCondition` пустую строку:

```
<add name="UrlRoutingModule-4.0" type="System.Web.Routing.UrlRoutingModule"
    preCondition="" />
```

Теперь перезапустите приложение в Visual Studio, чтобы измененные настройки вступили в силу, и перейдите по URL `/Content/StaticContent.html`. Вместо того чтобы увидеть содержимое файла, вы увидите сообщение об ошибке, показанное на рисунке 13-16. Эта ошибка возникла потому, поскольку запрос для HTML файла был передан системе маршрутизации MVC, но роут, совпадающий с URL, направляет запрос к контроллеру `Content`, который не существует.

**Рисунок 13-16:** Запрос файла со статическим контентом, который обрабатывается роутинговой системой



*Совет*

*Альтернативным подходом является использование сервера разработки Visual Studio, который вы можете активировать в разделе Web конфигурации проекта, если вы выберете при выборе пункт `UrlsAndRoutes Properties` из Visual Studio меню `Project`. Сервер разработки довольно прост и не является урезанной версией IIS, как IIS Express, и поэтому он не перехватывает запросы таким же образом.*

## Определение роутов для дисковых файлов

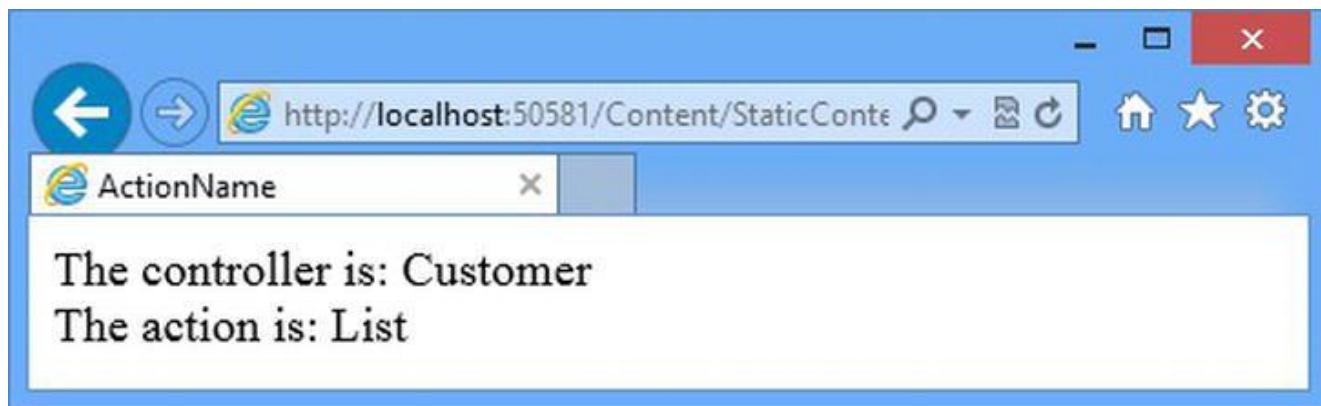
Как только свойство было установлено на `true`, мы можем определить роуты, подходящие URL, которые соответствуют дисковым файлам, как показано в листинге 13-34.

**Листинг 13-34:** Роут, чей URL паттерн соответствует дисковому файлу

```
public static void RegisterRoutes(RouteCollection routes) {
    routes.RouteExistingFiles = true;
    routes.MapRoute("DiskFile", "Content/StaticContent.html",
        new {
            controller = "Customer",
            action = "List",
        });
    routes.MapRoute("ChromeRoute", "{*catchall}",
        new { controller = "Home", action = "Index" },
        new {
            customConstraint = new UserAgentConstraint("Chrome")
        },
        new[] { "UrlsAndRoutes.AdditionalControllers" });
    routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{*catchall}",
        new { controller = "Home", action = "Index",
              id = UrlParameter.Optional },
        new[] { "URLsAndRoutes.Controllers" });
}
```

Этот роут направляет запросы для URL Content/StaticContent.html действию List контроллера Customer. Вы можете увидеть, как это работает на рисунке 13-17, которые мы создали, запустив приложение и перейдя снова по URL /Content/StaticContent.html.

**Рисунок 13-17:** Перехват запроса для дискового файла при помощи роута



Роутинговые запросы, предназначенные для файлов на диске, требуют тщательного анализа, не в последнюю очередь потому, что URL паттерны будут соответствовать этим видам URL так же хорошо, как и любым другим. Например, как было показано в предыдущем разделе, для /Content/StaticContent.html будет соответствовать URL паттерн {controller}/{action}. Если вы не будете осторожны, вы можете в конечном итоге получить исключительно странные результаты и снижение производительности. Эту опцию стоит включать только в крайнем случае.

## Обход системы маршрутизации

Настройка свойства RouteExistingFiles, которое мы показали в предыдущем разделе, делает систему маршрутизации более содержательной. Запросы, которые обычно обходят систему маршрутизации, в настоящее время оцениваются по отношению к нашим роутам, которые мы определили.

Противоположностью этой функции является возможность сделать систему маршрутизации менее содержательной, то есть ссылки не будут оцениваться по отношению к нашим роутам. Мы делаем это с помощью метода IgnoreRoute класса RouteCollection, как показано в листинге 13-35.

### Листинг 13-35: Использование метода IgnoreRoute

```
public static void RegisterRoutes(RouteCollection routes) {
    routes.RouteExistingFiles = true;
    routes.IgnoreRoute("Content/{filename}.html");
    routes.MapRoute("DiskFile", "Content/StaticContent.html",
        new {
            controller = "Customer",
            action = "List",
        });
    routes.MapRoute("ChromeRoute", "{*catchall}",
        new { controller = "Home", action = "Index" },
        new {
            customConstraint = new UserAgentConstraint("Chrome")
        },
        new[] { "UrlsAndRoutes.AdditionalControllers" });
    routes.MapRoute("MyRoute", "{controller}/{action}/{id}/{*catchall}",
        new { controller = "Home", action = "Index",
              id = UrlParameter.Optional },
        new[] { "URLsAndRoutes.Controllers" });
}
```

Мы можем использовать сегментные переменные, такие как `{filename}`, для соответствия с рядом URL. В этом случае URL паттерн будет соответствовать любому двухсегментному URL, где первым сегментом является Content, а второй сегмент имеет расширение .html.

Метод `IgnoreRoute` создает запись в `RouteCollection`, где обработчиком роута является экземпляр класса `StopRoutingHandler`, а не `MvcRouteHandler`. Система маршрутизации очень хорошо распознает этот обработчик. Если URL паттерн, переданный методу `IgnoreRoute`, находит соответствие, то никакие последующие роуты не будут оцениваться, как и в случае с обычным роутом. Отсюда следует, что важно то место, где вызывается метод `IgnoreRoute`. Если вы запустите приложение и снова перейдете по URL /Content/StaticContent.html, вы увидите содержимое HTML файла, так как объект `StopRoutingHandler` обрабатывается перед любым другим роутом.

## Резюме

В этой главе хорошо рассмотрели систему маршрутизации. Вы увидели, как подбираются и обрабатываются входящие запросы, как настроить роуты, изменив способ, которым они соответствуют URL сегментам, используя значения по умолчанию и дополнительные сегменты. Мы также показали вам, как ограничить роуты, чтобы сузить круг запросов, которым они будут соответствовать, и как направлять запросы для статического контента.

В следующей главе мы покажем вам, как генерировать исходящие URL из роутов в ваших представлениях и как использовать MVC области – возможность, которая опирается на систему маршрутизации и которая может быть использована для управления большими и сложными MVC приложениями.

# Расширенные функции маршрутизации

В предыдущей главе мы показали вам, как использовать систему маршрутизации для обработки входящих запросов, но это только часть истории. Мы также должны быть в состоянии использовать наши URL схемы для создания *исходящих* URL, которые мы можем встроить в наши представления, так чтобы пользователи могли щелкать по ссылкам и отправлять формы обратно в наше приложение способом, который будет направлен на правильный контроллер и действие. В этой главе мы покажем вам различные методы для генерации исходящих URL, расскажем, как настроить систему маршрутизации, заменяя стандартные MVC классы реализации маршрутизации и используя MVC *области* (*areas*), которые позволяют разбивать большие и сложные MVC приложения на управляемые части. Мы закончим эту главу некоторыми лучшими советами для URL схем в приложениях MVC фреймворка.

## Подготовка проекта для примера

Мы собираемся продолжать использовать проект из предыдущей главы `UrlsAndRoutes`, но мы должны сделать несколько изменений, прежде чем начать.

Сначала удалите папку `AdditionalControllers` и файл `HomeController.cs`, который в ней содержится. Мы создали эту папку и файл, чтобы продемонстрировать, как расставить приоритеты для пространств имен, и нам они больше не нужны. Для выполнения удаления, щелкните правой кнопкой мыши по папке `AdditionalControllers` и выберите `Delete` из всплывающего меню.

Другое изменение, которые мы должны сделать, заключается в том, чтобы упростить роуты в приложении. Отредактируйте файл `App_Start\RouteConfig.cs`, чтобы он соответствовал содержанию, показанному в листинге 14-1.

**Листинг 14-1:** Упрощение роутов в файле `RouteConfig.cs`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
using UrlsAndRoutes.Infrastructure;
namespace UrlsAndRoutes
{
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.MapRoute("MyRoute", "{controller}/{action}/{id}",
                new
                {
                    controller = "Home",
                    action = "Index",
                    id = UrlParameter.Optional
                });
        }
    }
}
```

# Создание исходящих URL в представлениях

Почти в каждом MVC приложении вы хотите позволить пользователю перемещаться от одного представления к другому, и это, как правило, основывается на том, что ссылка включается в первое представление и нацеливается на метод действия, который генерирует второе представление.

Так заманчиво просто добавить статический элемент `a`, атрибут `href` которого нацелен на метод действия:

```
<a href="/Home/CustomVariable">This is an outgoing URL</a>
```

Этот HTML элемент создает ссылку, которая будет обработана как запрос для метода действия `CustomVariable` контроллера `Home`, с дополнительной переменной сегмента `Hello`. Определенные вручную URL, как этот, быстро и просто создавать. Они также чрезвычайно опасны, и вам нужно будет ломать все жестко закодированные URL при изменении URL схемы вашего приложения. Затем вам нужно будет пройти по всем представлениям в приложении и обновить все ссылки для контроллеров и методов действий: это утомительный, подверженный ошибкам и плохо тестируемый процесс.

Гораздо лучшим вариантом является использование системы маршрутизации для генерации исходящих URL, которая гарантирует, что URL схема используется для получения URL динамически, таким способом, который гарантированно отражает URL схему приложения.

## Использование роутинговой системы для генерации исходящих URL

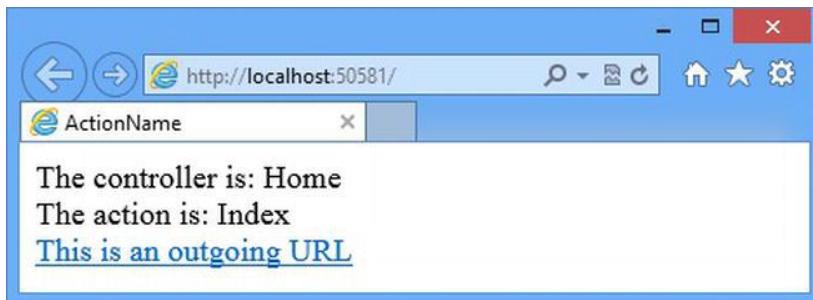
Самый простой способ генерации исходящих URL в представлении заключается в вызове вспомогательного метода `Html.ActionLink`, как показано на листинге 14-2, где продемонстрированы дополнения, которые мы внесли в файл представления `/Views/Shared/ActionName.cshtml`.

**Листинг 14-2:** Использование вспомогательного метода `Html.ActionLink` для генерации исходящих URL

```
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>ActionName</title>
</head>
<body>
    <div>The controller is: @ViewBag.Controller</div>
    <div>The action is: @ViewBag.Action</div>
    <div>
        @Html.ActionLink("This is an outgoing URL", "CustomVariable")
    </div>
</body>
</html>
```

Параметрами метода `ActionLink` являются текст для ссылки и имя метода действия, на который должна быть нацелена ссылка. Вы можете увидеть результат этого дополнения, запустив приложение и позволив браузеру перейти в корневой URL, как показано на рисунке 14-1.

**Рисунок 14-1:** Добавление в представление исходящего URL



HTML, который генерирует метод `ActionLink`, основывается на текущей конфигурации маршрутизации. Например, при использовании схемы, определенной в листинге 14-1 (и предполагая, что представление создается запросом к контроллеру `Home`), мы получаем этот HTML:

```
<a href="/Home/CustomVariable">This is an outgoing URL</a>
```

Может показаться, что мы прошли длинный путь, чтобы воссоздать определенный вручную URL, который мы показали вам раньше, но преимущество такого подхода заключается в том, что он автоматически реагирует на изменение конфигурации маршрутизации. В качестве демонстрации мы добавили новый роут в файл `RouteConfig.cs`, как показано в листинге 14-3.

**Листинг 14-3:** Добавление роута в приложение

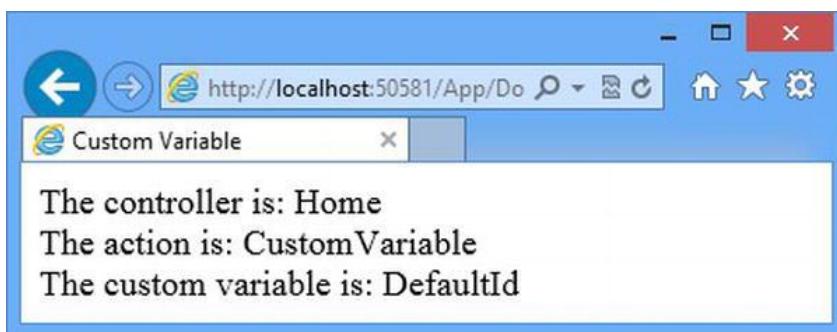
```
public static void RegisterRoutes(RouteCollection routes) {  
    routes.MapRoute("NewRoute", "App/Do{action}",  
        new { controller = "Home" });  
    routes.MapRoute("MyRoute", "{controller}/{action}/{id}",  
        new { controller = "Home", action = "Index",  
            id = UrlParameter.Optional });  
}
```

Новый роут меняет URL схему для запросов, нацеленных на контроллер `Home`. Если вы запустите приложение, вы увидите, что это изменение будет отражено в HTML, который создается вспомогательным методом `ActionLink`:

```
<a href="/App/DoCustomVariable">This is an outgoing URL</a>
```

Вы видите, как генерация ссылок таким образом решает вопрос о поддержке. Мы можем изменить нашу схему маршрутизации, и исходящие ссылки в наших представлениях автоматически отражают изменения. И, конечно исходящий URL становится регулярным запросом, когда вы нажимаете на ссылку, и поэтому система маршрутизации снова используется для того, чтобы ссылка была нацелена на нужный метод действия, как показано на рисунке 14-2.

**Рисунок 14-2:** Результат нажатия на ссылку заключается в том, чтобы сделать исходящий URL входящим запросом



## Понимание того, как выбираются роуты для генерации URL

Вы видели, как изменение роутов, которые определяют URL схему, меняет способ, которым создаются исходящие URL. Приложение, как правило, определяют несколько роутов, и важно понимать, как роуты выбираются для генерирования URL. Система маршрутизации обрабатывает роуты в том порядке, в котором они были добавлены в объект `RouteCollection`, который передается методу `RegisterRoutes`. Каждый роут проверяется на соответствие, и для этого должны быть выполнены три условия:

- Значение должно быть доступно для каждой сегментной переменной, определенной в URL паттерне. Чтобы найти значения для каждой сегментной переменной, система маршрутизации прежде всего рассматривает значения, которые предоставили мы (используя свойства анонимного типа), затем значения переменных для текущего запроса, и, наконец, значения по умолчанию, определенные в роуте. (Мы вернемся ко второму виду этих значений далее в этой главе).
- Ни одно из значений, которые мы предоставили для сегментных переменных, не может конфликтовать с переменными «только по умолчанию» (*default-only*), определенными в роуте. Это переменные, для которых были предоставлены значения по умолчанию, но которые не появляются в URL паттерне. Например, в этом определении роута переменная `myVar` является *default-only*:

```
routes.MapRoute("MyRoute", "{controller}/{action}",
    new { myVar = "true" } );
```

Чтобы для этого роута было соответствие, мы должны позаботиться о том, чтобы не указать значение `myVar`, или убедиться, что значение, которое мы предоставили, соответствует значению по умолчанию.

- Значения для всех сегментных переменных должны удовлетворять ограничению роутов. См. «Ограничение роутов» в предыдущей главе для информации по различным видам ограничений.

Чтобы внести ясность: система маршрутизации не пытается найти роут, который обеспечивает *наиболее подходящее* совпадение. Она находит только *первое* совпадение, и в этот момент она использует роут для генерации URL; любые последующие роуты игнорируются. По этой причине вы должны определить первыми наиболее конкретные роуты. Важно также тестировать генерирование исходящих URL. Если вы попытаетесь создать URL, для которых не может быть найден соответствующий роут, вы создадите ссылку, которая содержит пустой атрибут `href`:

```
<a href="">About this application</a>
```

Ссылка будет показывать представление должным образом, но не будет функционировать, как нужно, когда пользователь нажмет на нее. Если вы создаете только URL (мы покажем вам, как это делается, далее в этой главе), то результат будет `null`, который отображается в представлениях как пустая строка. Вы можете контролировать соответствие роутов с помощью именованных роутов. См. раздел «Создание URL из конкретного роута» далее в этой главе.

Первый объект `Route`, отвечающий этим критериям, создаст не-`null` URL, и это завершит процесс генерирования URL. Выбранные значения параметров будут заменены для каждого параметра сегмента. Если вы установили явные параметры, которые не соответствуют параметрам сегмента или параметрам по умолчанию, тогда метод будет добавлять их в виде набора пар имя/значение строки запроса.

## Юнит тест: тестирование исходящих URL

Самый простой способ проверить генерацию исходящего URL – то использовать статический метод `UrlHelper.GenerateUrl`, который имеет параметры для всех способов, которыми вы можете направлять генерирование роута, например, указав имя роута, контроллер, действие, значения сегментов и так далее. Вот тестовый метод, который проверяет генерирование URL по отношению к роуту, определенному в листинге 14-3. Мы добавили его в файл `RouteTests.cs` тестового проекта (потому что он использует тестовые методы, которые мы определили в предыдущей главе):

```
[TestMethod]
public void TestIncomingRoutes() {
    // ...код удален, чтобы предотвратить ошибки теста...
}

[TestMethod]
public void TestOutgoingRoutes() {
    // Arrange
    RouteCollection routes = new RouteCollection();
    RouteConfig.RegisterRoutes(routes);
    RequestContext context = new RequestContext(CreateHttpContext(), new RouteData());
    // Act – сгенерировать URL
    string result = UrlHelper.GenerateUrl(null, "Index", "Home", null,
        routes, context, true);
    // Assert
    Assert.AreEqual("/App/DoIndex", result);
}

...
...
```

Мы генерируем URL, а не ссылку, потому нам не нужно беспокоиться о тестировании окружающего HTML. Метод `UrlHelper.GenerateUrl` требует объект `RequestContext`, который мы создаем с помощью `mock-объекта HttpContextBase` вспомогательного метода тестирования `CreateHttpContext`. Прочтайте о юнит тестировании входящих URL в предыдущей главе, чтобы увидеть полный исходный код `CreateHttpContext`.

Мы удалили кода из метода `TestIncomingRoutes`, потому что наш тестовый код не отражается на изменениях, которые мы внесли в роуты в этой главе, и мы не хотим постоянно обновлять тест, поскольку мы нацелены на исходящие роуты.

## Нацилленность на другие контроллеры

Версия по умолчанию метода `ActionLink` предполагает, что вы хотите работать с методом действия в том же контроллере, который вызвал отображение представления. Чтобы создать исходящий URL, нацеленный на другой контроллер, вы можете использовать другой перегруженный вариант, который позволяет указать имя контроллера, как показано в листинге 14-4. Здесь мы внесли изменение в представление `ActionName.cshtml`.

### **Внимание**

*Система маршрутизации не имеет представления о нашем приложении при генерации исходящих URL. Это означает, что значения, которые вы передаете для методов действий и контроллеров, не проверяются, и вы должны позаботиться о том, чтобы не указать несуществующий целей.*

**Листинг 14-4:** Нацилленность на другой контроллер при помощи вспомогательного метода `ActionLink`

```
@{
    Layout = null;
}
<!DOCTYPE html>
```

```

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>ActionName</title>
</head>
<body>
    <div>The controller is: @ViewBag.Controller</div>
    <div>The action is: @ViewBag.Action</div>
    <div>
        @Html.ActionLink("This is an outgoing URL", "CustomVariable")
    </div>
    <div>
        @Html.ActionLink("This targets another controller", "Index", "Admin")
    </div>
</body>
</html>

```

Когда вы отобразите представление, вы увидите следующий сгенерированный HTML:

```
<a href="/Admin">This targets another controller</a>
```

Наш запрос для URL, предназначенного для метода действия `Index` контроллера `Admin`, был выражен методом `ActionLink` как `/Admin`. Система маршрутизации довольно умна, и она знает, что роут, определенный в приложении, будет использовать по умолчанию метод действия `Index`, что позволяет опускать ненужные сегменты.

## Передача особых значений

Вы можете передать значения для сегментных переменных, используя анонимный тип, при помощи свойств, представляющих сегменты. В листинге 14-5 приведен пример, который мы добавили в файл представления `ActionName.cshtml`.

### Листинг 14-5: Предоставление значений для сегментных переменных

```

<body>
    <div>The controller is: @ViewBag.Controller</div>
    <div>The action is: @ViewBag.Action</div>
    <div>
        @Html.ActionLink("This is an outgoing URL",
            "CustomVariable", new { id = "Hello" })
    </div>
</body>

```

В этом примере мы предоставили значение для сегментной переменной `id`. Если наше приложение использует роут, показанный в листинге 14-3, то мы получим следующий HTML, когда отобразим представление:

```
<a href="/App/DoCustomVariable?id=Hello">This is an outgoing URL</a>
```

Обратите внимание, что значение, которое мы передали, было добавлено в качестве части строки запроса, чтобы вписаться в URL паттерн, описанный роутом, который мы добавили в листинге 14-3. Это потому что нет никакой сегментной переменной, которая соответствует `id` в этом роуте. В листинге 14-6 мы отредактировали роуты в файле `RouteConfig.cs`, чтобы использовался лишь тот роут, который имеет сегмент `id`.

### Листинг 14-6: Отключение роута

```

...
public static void RegisterRoutes(RouteCollection routes) {

```

```
// Это выражения под комментариями
//routes.MapRoute("NewRoute", "App/Do{action}",
//    new { controller = "Home" });
routes.MapRoute("MyRoute", "{controller}/{action}/{id}",
    new { controller = "Home", action = "Index", id = UrlParameter.Optional });
}
...
...
```

Если мы запустим приложение еще раз, URL в представлении `ActionName.cshtml` создаст следующий HTML элемент:

```
<a href="/Home/CustomVariable>Hello">This is an outgoing URL</a>
```

На этот раз значение, которое мы назначили свойству `id`, включено в качестве URL сегмента, в соответствии с активным роутом в конфигурации приложения. Мы ведь говорили, что система маршрутизации довольно умна?

## Повторное использование сегментных переменных

Когда мы описывали способ, которым роуты сопоставляются с исходящими URL, мы объяснили, что при попытке найти значения для каждой сегментной переменной в URL паттерне роута система маршрутизации будет смотреть на значения из текущего запроса. Это поведение, которое смущает многих программистов и может привести к длительной отладке.

Представьте, что у нашего приложения есть единственный роут:

```
routes.MapRoute("MyRoute", "{controller}/{action}/{color}/{page}");
```

Теперь представьте, что пользователь на данный момент переходит по URL `/Catalog/List/Purple/123`, и мы отображаем ссылку следующим образом:

```
@Html.ActionLink("Click me", "List", "Catalog", new {page=789}, null)
```

Можно было бы ожидать, что система маршрутизации не в состоянии найти соответствие с роутом, потому что мы не предоставили значение переменной сегмента `color`, и для нее нет значения по умолчанию. Однако, это не так. Система маршрутизации найдет соответствие с роутом, который мы определили. Она генерирует следующий HTML:

```
<a href="/Catalog>List/Purple/789">Click me</a>
```

Система маршрутизации стремится найти соответствие с роутом, даже если ей придется заново использовать сегментную переменную из входящего URL. В данном случае мы в конечном итоге получаем для переменной `color` значение `Purple`, благодаря URL, с которого начал наш воображаемый пользователь.

Это нестина в последней инстанции. Система маршрутизации будет применять эту технику как часть своей регулярной оценки роутов, даже если есть последующие роуты, которые соответствовали бы URL, не требуя того, чтобы значения из текущего запроса были заново использованы. Система маршрутизации будет повторно использовать значения только для сегментных переменных, которые появляются раньше в URL паттерне, чем любые параметры, которые передаются методу `Html.ActionLink`. Предположим, что мы попытались создать ссылку следующим образом:

```
@Html.ActionLink("Click me", "List", "Catalog", new {color="Aqua"}, null)
```

Мы предоставили значение для `color`, но не для `page`. Но `color` появляется перед `page` в URL паттерне, и поэтому система маршрутизации не будет повторно использовать значения из входящего URL, и роут не будет совпадать.

Лучший способ борьбы с таким поведением – это не дать ему случиться. Мы настоятельно рекомендуем вам не полагаться на такое поведение, а также чтобы вы задавали значения для всех переменных сегмента в URL паттерне. Такое поведение не только сделает ваш код трудно читаемым, но вы в конечном итоге только и будете что сидеть и думать о том, каким же образом ваши пользователи делают запросы, что приведет к неоправданным времененным и трудовыми расходами.

## Указание HTML атрибутов

Мы сосредоточили свое внимание на URL, которые генерирует вспомогательный метод `ActionLink`, но помните, что метод создает полный якорь HTML элемента (`a`). Мы можем установить атрибуты для этого элемента, предоставляя анонимный тип, свойства которого соответствуют требуемым атрибутам. Листинг 14-7 показывает, как мы изменили представление `ActionName.cshtml` с целью установить атрибут `id` и присвоить класс HTML элементу `a`.

### Листинг 14-7: Генерирование якорного элемента с атрибутами

```
<body>
    <div>The controller is: @ViewBag.Controller</div>
    <div>The action is: @ViewBag.Action</div>
    <div>
        @Html.ActionLink("This is an outgoing URL",
                        "Index", "Home", null, new { id = "myAnchorID", @class = "myCSSClass" })
    </div>
</body>
```

Мы создали новый анонимный тип, который имеет свойства `id` и `class`, и передали его в качестве параметра методу `ActionLink`. Мы передали `null` для дополнительных значений сегментных переменных, указывая, что у нас нет никаких значений, которые мы можем предоставить.

#### *Совет*

*Обратите внимание, что мы поставили перед свойством `class` символ `@`. Это возможность языка C#, которая позволяет нам использовать зарезервированные ключевые слова в качестве имен членов класса.*

Когда отображается `ActionLink`, мы получаем следующий HTML:

```
<a class="myCSSClass" href="/" id="myAnchorID">This is an outgoing URL</a>
```

## Создание полных URL в ссылках

Все ссылки, которые мы генерировали, содержали относительные URL, но мы также можем использовать вспомогательный метод `ActionLink`, чтобы генерировать полные (абсолютные) ссылки, как показано в листинге 14-8.

### Листинг 14-8: Создание абсолютного URL

```
<body>
    <div>The controller is: @ViewBag.Controller</div>
    <div>The action is: @ViewBag.Action</div>
    <div>
        @Html.ActionLink("This is an outgoing URL", "Index", "Home",
                        new { id = "myAnchorID", @class = "myCSSClass" })
    </div>
</body>
```

```

    "https", "myserver.mydomain.com", "myFragmentName",
    new { id = "MyId" },
    new { id = "myAnchorID", @class = "myCSSClass" })
</div>
</body>

```

Это перегруженный метод `ActionLink` с большинством параметров, что позволяет нам предоставлять значения для протокола (`https`, в нашем примере), имя целевого сервера (`myserver.mydomain.com`) и URL фрагмент (`myFragmentName`), а также все другие опции, которые вы видели раньше. При отображении в представлении вызов метода генерирует следующий HTML:

```

<a class="myCSSClass"
    href="https://myserver.mydomain.com/Home/Index/MyId#myFragmentName"
    id="myAnchorID">This is an outgoing URL</a>

```

Мы рекомендуем использовать относительные URL везде, где это возможно. Полные URL создают зависимости для того, как инфраструктура вашего приложения представлена для пользователей. Мы видели много больших приложений, которые полагались на абсолютные URL, которые ломались из-за несогласованных изменений в сетевой инфраструктуре или политики доменных имен, что часто находится за пределами контроля программистов.

## Генерация URL (а не ссылок)

Вспомогательный метод `Html.ActionLink` создает полные HTML элементы `<a>`, а это именно то, чего мы хотим практически всегда. Тем не менее, бывают случаи, когда нам просто нужен URL. Это может случиться, потому что мы хотим отобразить URL, построить HTML для ссылки вручную, отобразить значение URL или включить URL в качестве элемента данных в HTML страницу, подлежащую отображению.

При таких обстоятельствах мы можем использовать метод `Url.Action` для создания только URL, а не окружающего HTML. Листинг 14-9 показывает изменения, которые мы внесли в файл `ActionName.cshtml`, чтобы создать URL при помощи `Url.Action`.

### Листинг 14-9: Создание URL без окружающего HTML

```

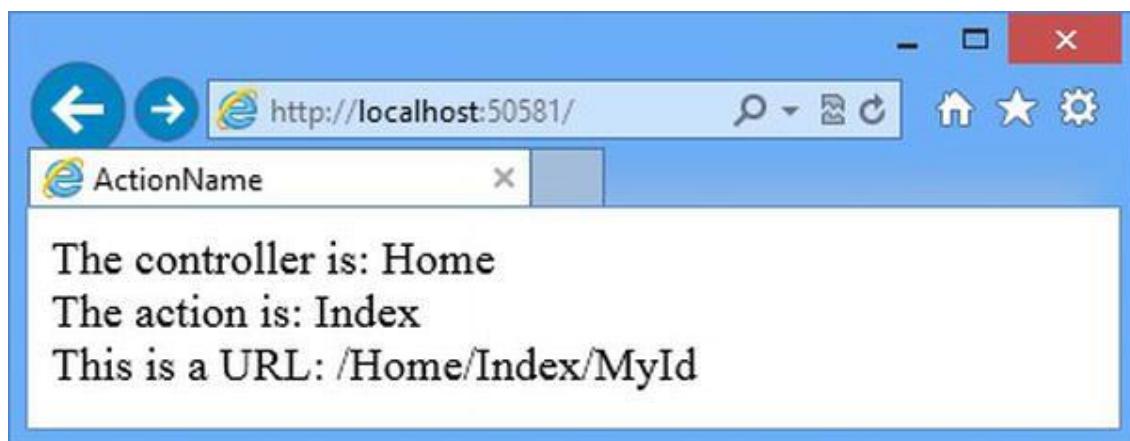
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>ActionName</title>
</head>
<body>
    <div>The controller is: @ViewBag.Controller</div>
    <div>The action is: @ViewBag.Action</div>
    <div>
        This is a URL:
        @Url.Action("Index", "Home", new { id = "MyId" })
    </div>
</body>
</html>

```

Метод `Url.Action` работает таким же образом, как и метод `Html.ActionLink`, за исключением того, что он генерирует только URL. Перегруженные версии метода и параметры, которые они принимают, одинаковы для обоих методов, и вы можете сделать все то же самое с `Url.Action`, что

мы продемонстрировали для `Html.ActionLink` в предыдущих разделах. На рисунке 14-3 показано, как отображается URL из листинга 14-9.

**Рисунок 14-3:** Отображение URL в представлении



## Создание исходящих URL в методах действия

В большинстве случаев мы хотим генерировать исходящие URL в представлениях, но бывают случаи, когда мы хотим сделать нечто подобное внутри метода действия. Если нам просто нужно создать URL, мы можем использовать тот же вспомогательный метод, который мы использовали в представлении, как показано в листинге 14-10. Здесь продемонстрирован новый метод действия, который мы добавили в контроллер `Home`.

**Листинг 14-10:** Создание исходящего URL в методе действия

```
public ViewResult MyActionMethod() {  
    string myActionUrl = Url.Action("Index", new { id = "MyID" });  
    string myRouteUrl = Url.RouteUrl(new { controller = "Home", action = "Index" });  
    //... сделать что-то с URL...  
    return View();  
}
```

Для роутинга в нашем примере приложения переменной `myActionUrl` будет присвоено значение `/Home/Index/MyID`, а переменной `myRouteUrl` будет присвоено значение `/`, что согласуется с результатами, которые получаются из вызова этих вспомогательных методов в представлении.

Более общим требованием является перенаправление браузера клиента на другой URL. Мы можем сделать это, возвращая результат вызова метода `RedirectToAction`, как показано в листинге 14-11.

**Листинг 14-11:** Перенаправление на другое действие

```
public RedirectToRouteResult MyActionMethod() {  
    return RedirectToAction("Index");  
}
```

Результатом метода `RedirectToAction` является `RedirectToRouteResult`, который указывает MVC выполнять инструкцию перенаправления к URL, который будет вызывать определенное действие. Есть обычные перегруженные версии метода `RedirectToAction`, которые указывают контроллер и значения для сегмента переменных в генерируемых URL.

Если вы хотите сделать перенаправление, используя URL, сгенерированный только из свойств объекта, вы можете использовать метод `RedirectToRoute`, как показано в листинге 14-12.

### Листинг 14-12: Перенаправление на URL, который сгенерирован из свойств анонимного типа

```
public RedirectToRouteResult MyActionMethod() {  
    return RedirectToRoute(new {  
        controller = "Home",  
        action = "Index",  
        id = "MyID" });  
}
```

Этот метод также возвращает объект  `RedirectToRouteResult` и имеет тот же результат, что и вызов метода  `RedirectToAction`.

## Генерирование URL из конкретного роута

В наших предыдущих примерах система маршрутизации выбирала роут, который использовался для генерации URL или ссылки. В этом разделе мы покажем вам, как контролировать этот процесс и выбирать конкретные роуты. В листинге 14-13 изменили информацию в файле `RouteConfig.cs`, чтобы лучше продемонстрировать эту возможность.

### Листинг 14-13: Изменение роутинговой конфигурации

```
public static void RegisterRoutes(RouteCollection routes) {  
    routes.MapRoute("MyRoute", "{controller}/{action}");  
    routes.MapRoute("MyOtherRoute", "App/{action}", new { controller = "Home" });  
}
```

Мы указали имена для обоих роутов: `MyRoute` и `MyOtherRoute`. Есть две причины именования роутов:

- В качестве напоминания о целях роута
- Чтобы вы могли выбрать конкретный роут для генерации исходящего URL

Мы организовали роуты так, чтобы наименее специфичный появился первым в списке. Это означает, что если бы нам надо было создать такую ссылку с помощью метода `ActionLink`:

```
@Html.ActionLink("Click me", "Index", "Customer")
```

исходящая ссылка всегда будет генерироваться при помощи `MyRoute`:

```
<a href="/Customer/Index">Click me</a>
```

Вы можете переписать роутовое поведение по умолчанию при помощи метода `Html.RouteLink`, который позволяет вам указать, какой роут вы хотите использовать:

```
@Html.RouteLink("Click me", "MyOtherRoute", "Index", "Customer")
```

В результате этого ссылка, сгенерированная вспомогательным методом, выглядит вот так:

```
<a Length="8" href="/App/Index?Length=5">Click me</a>
```

В данном случае контроллер, который мы указали, `Customer`, переписан роутом, и ссылка вместо этого нацелена на контроллер `Home`.

## Проблема именованных роутов

Проблема использования именованных роутов для генерации исходящих URL заключается в том, что таким образом страдает разделение понятий, которое занимает центральное место в MVC

паттерне. При создании ссылки или URL в представлении или методе действия мы фокусируемся на действии и контроллере, к которым будет направлен пользователь, а не на формате URL, который будет использоваться. Давая знания о различных роутах представлениям или контроллерам, мы создаем зависимости, которых мы предпочли бы избегать.

Мы стараемся избегать именования роутов (указывая `null` для параметра имени роута). Мы предпочитаем использовать комментарии в коде, чтобы напоминать самим себе, для чего предназначен конкретный роут.

## Настройка системы маршрутизации

Вы видели, какой гибкой и настраиваемой является система маршрутизации, но если она не соответствует вашим требованиям, вы можете подстроить ее поведение под ваши нужды. В этом разделе мы покажем вам два способа, как сделать это.

### Создание пользовательской реализации RouteBase

Если вам не нравится способ, которым стандартные объекты `Route` соответствуют URL, или вы хотите реализовать что-то необычное, вы можете создать альтернативный класс из `RouteBase`. Это дает вам контроль над тем, как URL находят соответствие, как извлекаются параметры и как создаются исходящие URL.

Чтобы создать класс из `RouteBase`, вам нужно реализовать два метода:

- `GetRouteData(HttpContextBase httpContext)`: это механизм, при котором работает *соответствие входящих URL*. Фреймворк вызывает этот метод для каждой записи `RouteTable.Routes`, пока одна из них не вернет значение `не-null`.
- `GetVirtualPath(RequestContext requestContext, RouteValueDictionary values)`: Это механизм, при котором работает *генерация исходящих URL*. Фреймворк вызывает этот метод для каждой записи `RouteTable.Routes`, пока одна из них не вернет значение `не-null`.

Для демонстрации этого вида настройки мы собираемся создать класс `RouteBase`, который будет обрабатывать наследованные URL запросы. Представьте себе, что мы перенесли существующее приложение в MVC фреймворк, но некоторые пользователи уже создали закладки для URL или закодировали их в скриптах. Мы по-прежнему хотим поддерживать те старые URL. Мы могли бы справиться с этим с помощью обычной системы маршрутизации, но эта проблема представляет собой хороший пример для данного раздела.

Для начала нам нужно создать контроллер, который будет получать наследные запросы. Мы назвали наш контроллер `LegacyController`, и его содержимое показано в листинге 14-14.

#### Листинг 14-14: Класс LegacyController

```
using System.Web.Mvc;
namespaceUrlsAndRoutes.Controllers
{
    public class LegacyController : Controller
    {
        public ActionResult GetLegacyURL(string legacyURL)
        {
            return View((object)legacyURL);
        }
    }
}
```

В этом простом контроллере метод действия `GetLegacyURL` принимает параметр и передает его в качестве модели представления в представление. Если бы мы действительно реализовали этот контроллер, мы бы использовали этот метод для извлечения файлов, которые были запрошены, а сейчас мы просто собираемся показать URL в представлении.

#### *Совет*

*Заметьте, что мы передали параметр методу `View` в листинге 14-14. Одна из перегруженных версий метода `View` принимает строку, определяющую имя представления для отображения, и без добавления параметра это была бы перегруженная версия, которая заставляет компилятор C# думать, чего же мы хотим. Чтобы избежать этого, мы вызвали перегруженную версию, которая передает модель представления и использует представление по умолчанию. Мы также могли бы решить эту проблему, используя перегруженную версию, которая принимает и имя представления, и модель представления, но мы предпочитаем не создавать явных связей между методами действия и представлениями.*

Представление, которое мы связали с этим методом действия, называется `GetLegacyURL.cshtml`. Оно показано в листинге 14-15.

#### **Листинг 14-15:** Представление `GetLegacyURL`

```
@model string
{
    ViewBag.Title = "GetLegacyURL";
    Layout = null;
}
<h2>GetLegacyURL</h2>
The URL requested was: @Model
```

Еще раз, это очень просто. Мы хотим продемонстрировать пользовательское поведение роутов, так что мы не будем тратить время на создание сложных действий и представлений. Сейчас мы достигли точки, когда мы можем создать наш вариант `RouteBase`.

#### **Роутинг входящих URL**

Мы создали класс `LegacyRoute`, который мы поместили в папку `Infrastructure` (сюда мы помещаем классы поддержки, которые действительно не попадают в другие категории). Класс показан в листинге 14-16.

#### **Листинг 14-16:** Класс `LegacyRoute`

```
using System;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
namespaceUrlsAndRoutes.Infrastructure
{
    public class LegacyRoute : RouteBase
    {
        private string[] urls;
        public LegacyRoute(params string[] targetUrls)
        {
            urls = targetUrls;
        }
        public override RouteData GetRouteData(HttpContextBase httpContext)
```

```

    {
        RouteData result = null;
        string requestedURL =
            httpContext.Request.AppRelativeCurrentExecutionFilePath;
        if (urls.Contains(requestedURL, StringComparer.OrdinalIgnoreCase))
        {
            result = new RouteData(this, new MvcRouteHandler());
            result.Values.Add("controller", "Legacy");
            result.Values.Add("action", "GetLegacyURL");
            result.Values.Add("legacyURL", requestedURL);
        }
        return result;
    }
    public override VirtualPathData GetVirtualPath(RequestContext requestContext,
    RouteValueDictionary values)
    {
        return null;
    }
}
}

```

Конструктор этого класса принимает массив строк, где представлены отдельные URL, которые будет поддерживать этот роутовый класс. Мы их укажем, когда далее будем регистрировать роут. В этом листинге стоит отметить метод `GetRouteData`, который вызывает систему маршрутизации, чтобы увидеть, можем ли мы обработать входящий URL.

Если мы не можем обработать запрос, то мы просто возвращаем `null`, и система маршрутизации перейдет к следующему роуту в списке и повторит процесс. Если мы можем обработать запрос, нам необходимо вернуть экземпляр класса `RouteData`, содержащий значения для переменных `controller` и `action`, и все остальное, что мы хотим передать методу действия.

Когда мы создаем объект `RouteData`, мы должны передать в обработчик, с которым мы хотим работать, значения, которые мы генерируем. Мы собираемся использовать стандартный класс `MvcRouteHandler`, который и придает смысл значениям `controller` и `action`:

```
result = new RouteData(this, new MvcRouteHandler());
```

Для подавляющего большинства MVC приложений вам потребуется этот класс, так как он объединяет систему маршрутизации и модель контроллер/действие MVC приложения. Но вы можете заменить `MvcRouteHandler`, и мы покажем вам это в разделе «Создание пользовательского обработчика роутов» далее в этой главе.

В этой реализации мы готовы обработать любой запрос для URL, которые были переданы нашему конструктору. Когда мы получаем такой URL, мы добавляем жестко закодированные значения для контроллера и метода действия в объект `RouteValues`. Мы также добавляем свойство `legacyURL`. Заметьте, что имя этого свойства совпадает с именем параметра нашего метода действия, и это гарантирует, что значение, которое мы генерируем здесь, будет передано в метод действия с помощью параметра.

Последним шагом является регистрация нового роута, который использует наш вариант `RouteBase`. Вы можете увидеть, как это делается, в листинге 14-17, который показывает дополнение к файлу `RouteConfig.cs`.

#### Листинг 14-17: Регистрация пользовательской реализации `RouteBase`

```
public static void RegisterRoutes(RouteCollection routes) {
    routes.Add(new LegacyRoute(
        "~/articles/Windows_3.1_Overview.html",
```

```

        "~/old/.NET_1.0_Class_Library"));
routes.MapRoute("MyRoute", "{controller}/{action}");
routes.MapRoute("MyOtherRoute", "App/{action}", new { controller = "Home" });
}

```

Мы создаем новый экземпляр нашего класса `LegacyRoute` и передаем ему нужные URL. Затем мы добавляем объект в `RouteCollection` с помощью метода `Add`. Теперь, когда мы запустим приложение и запросим один из URL, которые мы определили, запрос будет обрабатываться нашим пользовательским классом и будет нацелен на наш контроллер, как показано на рисунке 14-4.

**Рисунок 14-4:** Роутинг запросов при помощи пользовательской реализации `RouteBase`



### Генерирование исходящих URL

Для поддержки создания исходящих URL мы должны реализовать метод `GetVirtualPath` в нашем классе `LegacyRoute`. Еще раз, если мы не в состоянии справиться с запросом, мы даем знать об этом системе маршрутизации, возвращая `null`. В противном случае мы возвращаем экземпляр класса `VirtualPathData`. Листинг 14-18 показывает нашу реализацию этого метода.

**Листинг 14-18:** Реализация метода `GetVirtualPath`

```

public override VirtualPathData GetVirtualPath(RequestContext requestContext,
RouteValueDictionary values)
{
    VirtualPathData result = null;
    if (values.ContainsKey("legacyURL") &&
        urls.Contains((string)values["legacyURL"], StringComparer.OrdinalIgnoreCase))
    {
        result = new VirtualPathData(this, new UrlHelper(requestContext)
            .Content((string)values["legacyURL"]).Substring(1));
    }
    return result;
}

```

Мы передали сегментные переменные и другую сопутствующую информацию, используя анонимные типы, а система маршрутизации преобразовала их в объекты `RouteValueDictionary`. Листинг 14-19 показывает дополнение к `ActionName.cshtml`, который генерирует исходящий URL с помощью нашего пользовательского роута.

**Листинг 14-19:** Создание исходящего URL при помощи пользовательского роута

```

@{
    Layout = null;
}

```

```

}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>ActionName</title>
</head>
<body>
    <div>The controller is: @ViewBag.Controller</div>
    <div>The action is: @ViewBag.Action</div>
    <div>
        This is a URL:
        @Html.ActionLink("Click me", "GetLegacyURL",
            new { legacyURL = "~/articles/Windows_3.1_Overview.html" })
    </div>
</body>
</html>

```

Когда это представление рендерится, вспомогательный метод `ActionLink` генерирует следующий HTML, как вы и могли ожидать:

```
<a href="/articles/Windows_3.1_Overview.html">Click me</a>
```

Анонимный тип, созданный при помощи `legacyURL`, преобразуется в класс `RouteValueDictionary`, который содержит ключ с тем же именем. В этом примере мы решаем, что мы можем работать с запросом для исходящего URL, если есть ключ с именем `legacyURL` и если его значение является одним из URL, переданным в конструктор. Мы могли бы это еще конкретизировать и проверить значения `controller` и `action`, но для простого примера этого достаточно.

Если мы получаем соответствие, мы создаем новый экземпляр `VirtualPathData`, передавая ему ссылку на текущий объект и исходящий URL. Мы использовали метод `Content` класса `UrlHelper` для преобразования относительного URL в тот, который может быть передан в браузер. Система маршрутизации добавляет дополнительный `/` в URL, поэтому мы должны позаботиться о том, чтобы удалить символ из нашего генерируемого URL.

## Создание пользовательского обработчика роутов

Мы работали с `MvcRouteHandler` в наших примерах, потому что он соединяет систему маршрутизации к MVC фреймворком. И поскольку наше внимание сосредоточено MVC, это нам фактически всегда подходит. Несмотря на это, система маршрутизации позволяет нам определять наш собственный обработчик роутов путем реализации интерфейса `IRouteHandler`. Листинг 14-20 показывает содержимое класса `CustomRouteHandler`, который мы добавили в папку `Infrastructure` нашего проекта.

### Листинг 14-20: Реализация интерфейса `IRouteHandler`

```

using System.Web;
using System.Web.Routing;
namespaceUrlsAndRoutes.Infrastructure
{
    public class CustomRouteHandler : IRouteHandler
    {
        public IHttpHandler GetHttpHandler(RequestContext requestContext)
        {
            return new CustomHttpHandler();
        }
    }
    public class CustomHttpHandler : IHttpHandler
    {

```

```

public bool IsReusable
{
    get { return false; }
}
public void ProcessRequest(HttpContext context)
{
    context.Response.Write("Hello");
}
}
}

```

Цель интерфейса `IRouteHandler` заключается в обеспечении средства для генерации реализаций интерфейса `IHttpHandler`, который отвечает за обработку запросов. При MVC реализации этих интерфейсов находятся контроллеры, вызываются методы действия, рендерятся представления и результаты записываются в ответ. Наша реализация немного проще. Она просто пишет клиенту слово `Hello` (не HTML документ, содержащий это слово, а только текст). Мы можем зарегистрировать наш пользовательский обработчик в файле `RouteConfig.cs`, когда мы определяем роут, как показано в листинге 14-21.

**Листинг 14-21:** Использование пользовательского обработчика роутов

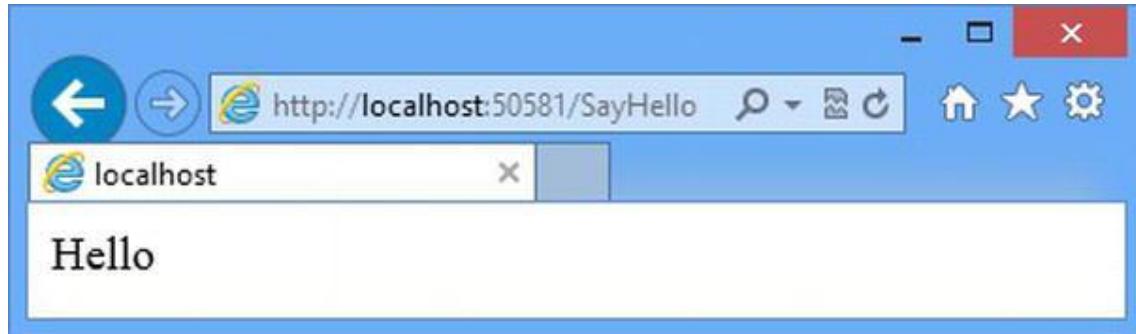
```

public static void RegisterRoutes(RouteCollection routes) {
    routes.Add(new Route("SayHello", new CustomRouteHandler()));
    routes.Add(new LegacyRoute(
        "~/articles/Windows_3.1_Overview.html",
        "~/old/.NET_1.0_Class_Library"));
    routes.MapRoute("MyRoute", "{controller}/{action}");
    routes.MapRoute("MyOtherRoute", "App/{action}", new { controller = "Home" });
}

```

Когда мы запрашиваем URL `/sayHello`, наш обработчик используется для обработки запроса. На рисунке 14-5 показан результат.

**Рисунок 14-5:** Использование пользовательского обработчика роутов



Реализация пользовательской обработки роутов означает то, что вы берете на себя ответственность за функции, которые обычно обрабатываются для вас, такие как реализация контроллеров и действий. Но это дает вам невероятную свободу. Вы можете объединять некоторые части MVC фреймворка и игнорировать другие или даже реализовать совершенно новый архитектурный паттерн.

## Лучшие примеры URL схемы

После всего этого вам, наверное, остается только гадать, с чего начать, чтобы создать собственную URL схему. Вы можете просто принять схему по умолчанию, которую Visual Studio генерирует для вас, но есть некоторые преимущества в создании чего-то своего.

В последние годы дизайн URL стали воспринимать довольно серьезно, и появилось несколько важных принципов проектирования. Если вы последуете этим паттернам проектирования, вы улучшите юзабилити и совместимость, а также поднимите поисковой рейтинг вашего приложения.

## Делайте свои URL чистыми и дружелюбными

Пользователи обращают внимание на URL в приложениях. Если вы не согласны, то просто вспомните последний раз, когда вы пытались послать кого-нибудь на URL от Amazon. Вот URL для этой книги:

[http://www.amazon.com/Pro-ASP-NET-MVC-Professional-Apress/dp/1430242361/ref=la\\_B001IU0SNK\\_1\\_5?ie=UTF8&qid=1349978167&sr=1-5](http://www.amazon.com/Pro-ASP-NET-MVC-Professional-Apress/dp/1430242361/ref=la_B001IU0SNK_1_5?ie=UTF8&qid=1349978167&sr=1-5)

Такую ссылку не особо приятно отправлять даже по электронной почте, но попробуйте прочитать это по телефону. Когда нам нужно было это сделать, мы просто посмотрели номер ISBN и попросили звонившего самого найти книгу. Было бы неплохо, если бы мы могли получить доступ к книге с URL вроде этого:

<http://www.amazon.com/books/pro-aspnet-mvc4-framework>

Это тот вид URL, который мы могли бы продиктовать по телефону, и он не выглядит так, как будто мы что-то уронили на клавиатуру при составлении сообщения электронной почты.

### Примечание

*Чтобы внести ясность, мы очень уважаем Amazon, который продает больше наших книг, чем все остальные, вместе взятые. Мы знаем, что каждый член команды Amazon является поразительно умным и красивым человеком. И никто из них не является мелочным, чтобы прекратить продажи наших книг из-за чего-то настолько незначительного, как критика их URL формата. Мы любим Amazon. Мы обожаем Amazon. Мы просто хотим, чтобы они поправили свои URL.*

Вот несколько советов по созданию дружелюбных URL:

- Создавайте URL, описывая их содержание, а не внося информацию о реализации вашего приложения. Используйте /Articles/AnnualReport, а не /Website\_v2/CachedContentServer/FromCache/AnnualReport.
- Предпочитайте названия номерам ID. Используйте /Articles/AnnualReport, а не /Articles/2392. Если вы должны использовать номер ID (чтобы различать элементы с одинаковыми названиями, или чтобы избежать дополнительных запросов к базе данных, необходимых для нахождения элемента по его названию), делайте это вот так: /Articles/2392/AnnualReport (используйте и название, и ID). Это займет больше времени, но это понятно пользователям и улучшает рейтинг в поисковых системах. Ваше приложение может просто игнорировать название и отображать элемент, соответствующий данному ID.
- Не используйте расширения имен файлов для HTML страниц (например, .aspx or .mvc), но используйте их для специализированных типов файлов (например, .jpg, .pdf и .zip). Веб-браузеры не сильно заботятся о расширениях имен файлов, если надлежащим образом установить MIME тип, но люди все еще ожидают PDF файлы на конце с .pdf.
- Придавайте смысловую иерархию (например, /Products/Menswear/Shirts/Red), чтобы ваш пользователь мог понять, какой URL относится к родительской категории.
- Не учитывайте регистр (пусть ваш URL будет регистронезависимым). Роутинговая система ASP.NET по умолчанию не учитывает регистр.

- Избегайте символов, кода и последовательности символов. Если вы хотите разделить слова, используйте дефисы (как в `/my-great-article`). Подчеркивания недружелюбны, а закодированные пробелы странные (`/my+great+article`) или отвратительные (`/my%20great%20article`).
- Не меняйте URL. Неработающие ссылки равны потере бизнеса. Если все же вы меняете URL, продолжайте поддерживать старую URL схему как можно дольше через перенаправление 301.
- Будьте последовательны. Придерживайтесь одного URL формата во всем приложении.

URL должны быть краткими, легко набираемыми, редактируемыми (чтобы пользователь мог сам работать с URL) и надежными, также они должны визуализировать структуру сайта. Якоб Нильсен, гуру юзабилити, развивает эту тему на <http://www.useit.com/alertbox/990321.html>. Тим Бернерс-Ли, изобретатель Интернета, дает подобный совет (см. <http://www.w3.org/Provider/Style/URI>).

## GET и POST: выберите правильный

Негласное правило гласит, что запросы GET следует использовать для получения информации «только для чтения» (read-only), в то время как POST запросы должны быть использованы для любой операции, которая изменяет состояние приложения. В терминах стандартизации, GET запросы служат для **безопасного взаимодействия** (только для получения информации), а POST запросы для **небезопасного взаимодействия** (принятие решения или изменение чего-то). Эти соглашения устанавливаются World Wide Web Consortium (W3C), на <http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html>.

GET запросы являются адресуемыми: вся информация содержится в URL, так что можно создавать закладки и делать ссылки на эти адреса.

Не используйте GET запросы для операций, изменяющих состояние. Многие веб-разработчики узнали это на собственном горьком опыте в 2005 году, когда был публично выпущен Google Web Accelerator. Это приложение прошло по всему контенту, для которого были назначены ссылки, что является законным для HTTP GET запросов, потому что они должны быть безопасными. К сожалению, многие веб-разработчики проигнорировали HTTP соглашение и разместили простые ссылки для "удалить" или "добавить в корзину" в своих приложениях. Начался хаос.

Одна из компаний посчитала, что их система управления контентом была целю повторяющихся атак, потому что все содержимое снова и снова удалялось. Позже они выяснили, что поисковой робот натолкнулся на URL административной страницы и прошел по всем ссылкам для удаления. Аутентификация может защитить вас от этого, но она не сможет защитить вас от поисковых роботов (краулеров).

## Резюме

В этой главе мы показали вам расширенные возможности системы маршрутизации MVC фреймворка, рассказали, как генерировать исходящие ссылки и URL и как настроить систему маршрутизации. Попутно мы ввели понятие области и изложили нашу позицию о том, как создать полезную и качественную URL схему.

В следующей главе мы обратимся к контроллерам и действиям, которые являются фактически сердцем MVC модели. Мы объясним, как они работают, и покажем вам, как их использовать, чтобы получить лучшие результаты для приложения.

# Контроллеры и методы действий

Каждый запрос, который приходит в ваше приложение, обрабатывается контроллером. Контроллер может свободно обработать любой подходящий запрос, пока он не войдет в зону ответственности модели и представления. Это означает, что мы не добавляем в контроллер бизнес логику или логику хранения данных, а также мы не создаем тут пользовательские интерфейсы.

В ASP.NET MVC фреймворке контроллеры – это .NET классы, содержащие логику, необходимую для обработки запроса. В главе 3 мы объяснили, что роль контроллера заключается в инкапсуляции логики приложения. Это обозначает, что контроллеры отвечают за обработку входящих запросов, выполняя операции по доменной модели и выбирая представления для отображения пользователю.

В этой главе мы покажем вам, как реализованы контроллеры, и представим различные способы, которыми вы можете использовать контроллеры, чтобы получать и генерировать выходные данные. MVC не ограничивает вас созданием HTML при помощи представлений, и мы обсудим тут другие доступные варианты. Мы также покажем, как методы действий облегчают выполнение модульного тестирования, и продемонстрируем, как проверять каждый вид результата методов действия.

## Знакомство с контроллером

Вы видели использование контроллеров практически всех предыдущих главах. Теперь пришло время сделать небольшой шаг назад и взглянуть за кулисы. Для начала нам нужно создать проект для наших примеров.

### Подготовка нашего проекта

Чтобы подготовиться к этой главе, мы создали новый MVC проект, который называется `ControllersAndActions`, с помощью шаблона `Empty`. Мы выбрали шаблон `Empty`, потому что мы собираемся создавать все контроллеры и представления, которые нам нужны по мере продвижения по главе.

#### *Совет*

*Не забудьте создать проект модульного теста, если вы хотите следовать примерам тестов, которые мы предлагаем в этой главе.*

### Создание контроллера при помощи `IController`

В MVC фреймворке классы контроллеров должны реализовывать интерфейс `IController` пространства имён `System.Web.Mvc`, как показано в листинге 15-1.

#### Листинг 15-1: Интерфейс `System.Web.Mvc.IController`

```
public interface IController
{
    void Execute(RequestContext requestContext);
}
```

Это очень простой интерфейс. Единственный метод, `Execute`, вызывается при запросе, который ориентирован на класс контроллера. MVC фреймворк знает, на какой класс был нацелен запрос, прочитав значение свойства `controller`, полученное от роутовых данных.

Вы можете выбрать для создания классов контроллеров реализацию `IController`, но это довольно низкоуровневый интерфейс, и вы должны проделать много работы, чтобы получить что-нибудь полезное. В листинге 15-2 показан простой контроллер `BasicController`, на основе которого представлена эта возможность.

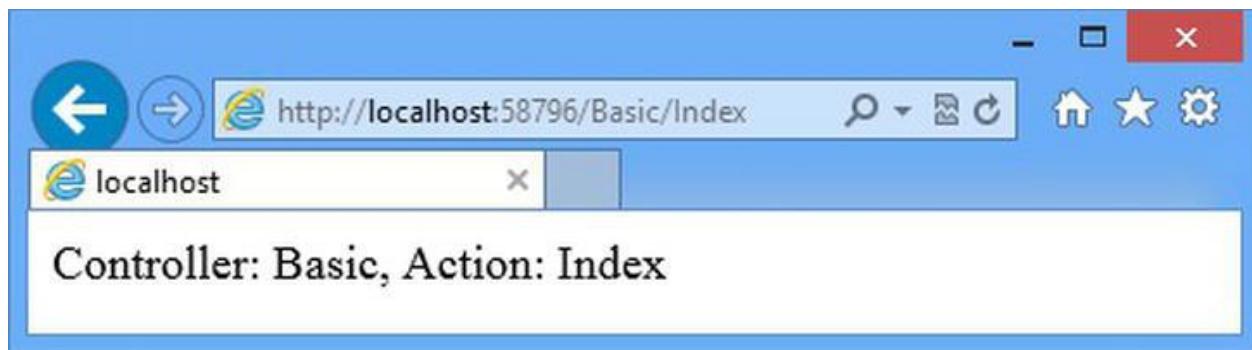
#### Листинг 15-2: Класс `BasicController`

```
using System.Web.Mvc;
using System.Web.Routing;
namespace ControllersAndActions.Controllers
{
    public class BasicController : IController
    {
        public void Execute(RequestContext requestContext)
        {
            string controller = (string)requestContext.RouteData.Values["controller"];
            string action = (string)requestContext.RouteData.Values["action"];
            requestContext.HttpContext.Response.Write(
                string.Format("Controller: {0}, Action: {1}", controller, action));
        }
    }
}
```

Для создания этого класса мы щелкнули правой кнопкой мыши по папке `Controllers` проекта и выбрали `Add -> New Class`. Затем мы назвали наш контроллер `BasicController` и добавили код, показанный в листинге 15-2.

В нашем методе `Execute` мы читаем значения переменных `controller` и `action` из объекта `RouteData`, связанного с запросом, и записываем их в результат. Если вы запустите приложение и перейдите по `/Basic/Index`, вы увидите результат выполнения нашего контроллера, как показано на рисунке 15-1.

Рисунок 15-1: Результат, сгенерированный классом `BasicController`



Реализация интерфейса `IController` позволяет создать класс, который MVC распознает в качестве контроллера и которому отправляет запросы, но тут довольно трудно написать сложное приложение. MVC не определяет, как контроллер работает с запросами, что обозначает, что вы можете создать свой желаемый способ.

#### Совет

*Обратите внимание, что MVC фреймворк не указывает вид движка для нашего базового контроллера. То, как создаются ответы, находится на усмотрении контроллера, и MVC фреймворк не делает никаких предположений о способе, который будет использоваться.*

## Создание контроллера путем наследования от класса Controller

MVC фреймворк бесконечно настраиваемый и расширяемый. Вы можете реализовать интерфейс `IController` для создания любого вида обработки запросов и генерации результата, который вам требуется. Не нравятся методы действий? Не важны отображаемые представления? Тогда вы можете просто взять дело в свои руки и написать лучший, более быстрый и более элегантный способ обработки запросов. Или же вы можете опираться на возможности, которые предоставила команда MVC, и это достигается наследованием ваших контроллеров от класса `System.Web.Mvc.Controller`.

`System.Web.Mvc.Controller` является классом, обеспечивающим поддержку обработки запросов, которая знакома большинству MVC разработчиков. Это то, что мы используем во всех наших примерах в предыдущих главах.

Класс `Controller` предоставляет три ключевые возможности:

- *Методы действия:* Поведение контроллера разделено на множество методов (вместо того, чтобы иметь только один метод `Execute()`). Каждый метод действия срабатывает для определенного, «своего» URL и вызывается с параметрами, извлеченными из входящего запроса.
- *Результаты действия:* Вы можете вернуть объект, описывая результат действия (например, отображение представления или перенаправление на другой URL или метод действия), который затем вы можете использовать по своему усмотрению. Разделение между *указанием результатов* и *их выполнением* упрощает модульное тестирование.
- *Фильтры:* Вы можете инкапсулировать повторяющиеся виды поведения (например, аутентификацию, как вы видели в главе 11) в качестве фильтров, а затем добавлять каждый вид поведения в один или несколько контроллеров или методов действия, разместив `[Attribute]` в исходном коде.

Пока вам не нужно реализовывать *очень* конкретные требования, лучшим способом для создания контроллеров является наследование от класса `Controller`, и, как вы знаете, это то, что делает Visual Studio, когда создает новый класс в ответ на `Add -> Controller`. В листинге 15-3 показан простой контроллер, созданный таким образом. Мы назвали наш класс `DerivedController`.

**Листинг 15-3:** Простой контроллер, унаследованный от класса `System.Web.Mvc.Controller`

```
using System.Web.Mvc;
namespace ControllersAndActions.Controllers
{
    public class DerivedController : Controller
    {
        public ActionResult Index()
        {
            ViewBag.Message = "Hello from the DerivedController Index method";
            return View("MyView");
        }
    }
}
```

Базовый класс `Controller` реализует метод действия `Execute` и отвечает за вызов метода действия, имя которого соответствует значению `action` в роутовых данных.

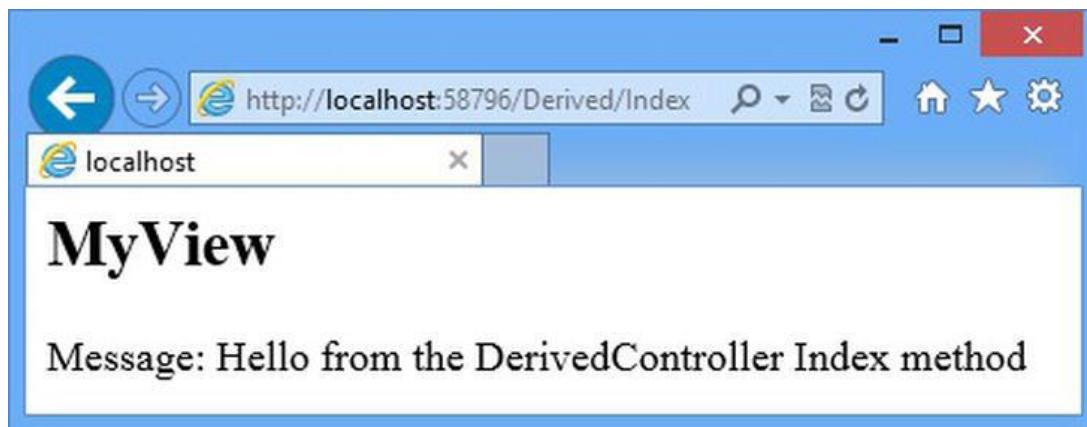
Класс `Controller` также является нашей связью с системой представления Razor. В листинге мы возвращаем результат метода `View`, передавая имя представления, которое мы хотим отобразить, клиенту в качестве параметра. Листинг 15-4 показывает это представление `MyView.cshtml`, которое расположено в папке `Views/Derived`.

#### Листинг 15-4: Файл MyView.cshtml

```
@{  
    ViewBag.Title = "MyView";  
}  
<h2>MyView</h2>  
Message: @ViewBag.Message
```

Если вы запустите приложение и перейдете на /Derived/Index, будет выполнен метод действия, который мы определили, и будет отображено нужное представление, как показано на рисунке 15-2.

**Рисунок 15-2:** Результат, сгенерированный классом DerivedController



Наша работа заключается в реализации методов действий, в получении всех входных данных для обработки запроса, а также создании подходящего ответа. Есть множество способов, которыми мы можем сделать это, и они рассматриваются в оставшейся части этой главы.

## Получение входных данных

Контроллерам часто необходим доступ к входящим данным, таким как значения строки запроса, значения форм и параметры, полученные из входящего URL системой маршрутизации. Есть три основных способа доступа к этим данным:

- Получить их из набора *контекстных объектов*
- Получить данные, переданные в качестве *параметров* методу действия
- Явно вызвать связывание данных модели

Здесь мы рассмотрим подходы получения входных данных для методов действия, уделяя особое внимание использованию контекстных объектов и параметров метода действия. В главе 22 мы подробно изучим связывание данных модели.

### Получение данных из контекстных объектов

Самый прямой способ получить данные заключается в том, чтобы извлечь их самостоятельно. Когда вы создаете контроллер путем наследования от базового класса `Controller`, вы получаете доступ к набору полезных свойств для получения информации о запросе. Эти свойства включают `Request`, `Response`, `RouteData`, `HttpContext` и `Server`. Каждое из них предоставляет информацию о различных аспектах запроса. Эти свойства получают различные типы данных из экземпляра запроса `ControllerContext` (который может быть доступным через свойство `Controller.ControllerContext`). Мы описали некоторые из наиболее часто используемых контекстных объектов в таблице 15-1.

**Таблица 15-1:** Наиболее часто используемые контекстные объекты

Свойство	Тип	Описание
Request.QueryString	NameValueCollection	Переменные GET, отправленные с этим запросом
Request.Form	NameValueCollection	Переменные POST, отправленные с этим запросом
Request.Cookies	HttpCookieCollection	Куки, отправленные браузером с этим запросом
Request.HttpMethod	string	HTTP метод (например, GET или POST), используемый для этого запроса
Request.Headers	NameValueCollection	Полный набор HTTP заголовков, отправленный с этим запросом
Request.Url	Uri	Запрашиваемый URL
Request.UserHostAddress	string	IP адрес пользователя, сделавшего запрос
RouteData.Route	RouteBase	Выбранная запись из RouteTable.Routes для этого запроса
RouteData.Values	RouteValueDictionary	Активные роутовые параметры (как полученные из URL, так и значения по умолчанию)
HttpContext.Application	HttpApplicationStateBase	Состояние приложения
HttpContext.Cache	Cache	Кэш приложения
HttpContext.Items	IDictionary	Состояние текущего запроса
HttpContext.Session	HttpSessionStateBase	Состояние сессии пользователя
User	IPrincipal	Информация об аутентификации залогиненного пользователя
TempData	TempDataDictionary	Информация о временных данных текущего пользователя

Метод действия может использовать любой из этих контекстных объектов, чтобы получить информацию о запросе, как показано в листинге 15-5, который демонстрирует гипотетический метод действия.

**Листинг 15-5:** Метод действия использует контекстные объекты, чтобы получить информацию о запросе

```
public ActionResult RenameProduct() {
    // Доступ к различным свойствам контекстных объектов
    string userName = User.Identity.Name;
    string serverName = Server.MachineName;
    string clientIP = Request.UserHostAddress;
    DateTime dateStamp = HttpContext.Timestamp;
    AuditRequest(userName, serverName, clientIP, dateStamp, "Renaming product");
    // Получение данных из Request.Form
    string oldProductName = Request.Form["OldName"];
    string newProductName = Request.Form["NewName"];
    bool result = AttemptProductRename(oldProductName, newProductName);
    ViewData["RenameResult"] = result;
    return View("ProductRenamed");
}
```

Вы можете просмотреть широкий спектр доступной контекстной информации о запросе при помощи IntelliSense (в методе действия наберите `this.` и просмотрите информацию во всплывающем окне) и Microsoft Developer Network (посмотрите `System.Web.Mvc.Controller` и его базовые классы или `System.Web.Mvc.ControllerContext`).

## Использование параметров метода действия

Как вы видели в предыдущих главах, методы действий могут принимать параметры. Это более аккуратный способ получения входящих данных, чем извлечение их вручную из контекстных объектов, и благодаря этому методы действий легче читать. Например, предположим, у нас есть метод действия, который использует контекстные объекты таким образом:

```
...
public ActionResult ShowWeatherForecast() {
    string city = (string)RouteData.Values["city"];
    DateTime forDate = DateTime.Parse(Request.Form["forDate"]);
    // ... здесь идет реализация прогноза погоды ...
    return View();
}
...
```

Мы можем переписать его, чтобы использовать параметры:

```
...
public ActionResult ShowWeatherForecast(string city, DateTime forDate) {
    // ... здесь идет реализация прогноза погоды ...
    return View();
}
```

Это не только легче читать, но это также помогает с модульным тестированием: мы можем проверить метод действия без необходимости использовать мок-технологию для свойств класса контроллера.

Для полноты картины стоит отметить, что методам действия не разрешается параметры `out` или `ref`. Это не имело бы никакого смысла, если бы разрешалось, и ASP.NET MVC просто выбросит исключение, если он увидит такие параметры.

MVC фреймворк предоставит значения для наших параметров, проверив контекстные объекты, в том числе `Request.QueryString`, `Request.Form` и `RouteData.Values`. Имена наших параметров рассматриваются как регистронезависимые, так что параметр метода действия `city` можно заполнить значением из `Request.Form["City"]`.

## Понимание того, как создаются экземпляры объектов параметров

Базовый класс `Controller` получает значения для параметров вашего метода действия с помощью MVC компонентов, называемых *провайдерами значений* и *механизмами связывания данных модели*.

Провайдеры значений представляют собой совокупность элементов данных, доступных для вашего контроллера. Есть встроенные провайдеры значений, которые получают элементы из `Request.Form`, `Request.QueryString`, `Request.Files` и `RouteData.Values`. Затем значения передаются механизмам связывания данных, которые пытаются привязать их к типам, которые методы действий требуют в качестве параметров.

Дефолтовые механизмы связывания данных могут создать и заполнить объекты любого .NET типа, в том числе коллекции и специфические пользовательские типы для конкретного проекта. Вы видели

пример этого в главе 10, когда сообщения от администраторов были представлены нашему методу действия как один объект `Product`, даже при том что отдельные значения были рассеяны среди элементов HTML формы. Мы расскажем о провайдерах значений и механизмах связывания данных модели в главе 22.

## Понимание факультативных и обязательных параметров

Если MVC фреймворк не может найти значение параметра ссылочного типа (например, `string` или `object`), все равно будет вызван метод действия, но с использованием значения `null` для этого параметра. Если значение не может быть найдено для параметра простого типа (например, `int` или `double`), то будет сгенерировано исключение, и метод действия не будет вызван. Вот почему это так:

- Параметры простого типа являются обязательными. Чтобы сделать их необязательными, либо укажите значение по умолчанию (см. следующий раздел), либо замените тип параметра на `nullable` (например, `int?` или `DateTime?`). Таким образом, MVC сможет передать `null`, если значение будет не доступно.
- Параметры ссылочного типа не являются обязательными. Чтобы сделать их обязательными (чтобы убедиться, что передается значение `не-null`), добавьте код в начало метода действия, который не принимает значения `null`. Например, если значение равно `null`, выбрасывается исключение `ArgumentNullException`.

## Указание значений параметров по умолчанию

Если вы хотите обрабатывать запросы, которые не содержат значений для параметров метода действия, но вы не хотите проверять в коде наличие значений `null` или выбрасывать исключения, вы можете использовать дополнительные параметры C#. В листинге 15-6 показан пример.

### Листинг 15-6: Использование дополнительных параметров C# для метода действия

```
...
public ActionResult Search(string query= "all", int page = 1) {
    // ...обработка запроса...
    return View();
}
...
```

Мы обозначаем параметры как дополнительные путем присвоения значений, когда мы их определяем. В листинге мы предоставили значения по умолчанию для параметров `query` и `page`. MVC Framework попытается получить значения из запроса для этих параметров, но если доступных значений не будет, будут использоваться значения по умолчанию, которые мы указали.

Для параметра `string query`, это означает, что нам не нужно проверять наличие значений `null`. Если в запросе, который мы обрабатываем, не задана строка запроса, тогда наш метод действия будет вызван со строкой `all`. Для параметра `int` нам не нужно беспокоиться о том, что запросы завершатся ошибками, если нет значения `page`. Наш метод действия будет вызываться со значением по умолчанию равным 1.

Необязательные параметры могут быть использованы для *литеральных типов*: это типы, которые можно определить без использования ключевого слова `new`, включая `string`, `int` и `double`.

### Внимание

*Если запрос содержит значение для параметра, но оно не может быть преобразовано в правильный тип (например, если пользователь дает нечисловую*

*строку для параметра `int`), то фреймворк передаст значение по умолчанию для этого типа параметра (например, 0 для параметра `int`) и зарегистрирует предоставленное значение как ошибку валидации в специальном контекстном объекте `ModelState`. Если вы не проверите ошибки валидации в `ModelState`, вы можете попасть в неприятную ситуацию, когда пользователь ввел неправильные данные в форму, но запрос обрабатывается, как если бы пользователь не ввел никаких данных вообще или ввел значение по умолчанию. См. главу 23 для информации о валидации и `ModelState`, который может быть использован, чтобы избежать таких проблем.*

## Создание выходных данных

После того как контроллер завершит обработку запроса, он обычно должен сгенерировать ответ. Когда мы создавали наш пустой контроллер напрямую путем реализации интерфейса `IController`, мы должны были взять на себя ответственность за все аспекты обработки запроса, в том числе за создание ответа клиенту. Если мы хотим, например, отправить HTML ответ, то мы должны создать и скомпоновать HTML данные и отправить их клиенту, используя метод `Response.Write`. Аналогичным образом, если мы хотим перенаправить браузер пользователя на другой URL, мы должны вызвать метод `Response.Redirect` и передать интересующий нас URL напрямую. Оба этих подхода показаны в листинге 15-7, который демонстрирует улучшения в классе `BasicController`.

**Листинг 15-7:** Генерирование результатов в реализации `IController`

```
using System.Web.Mvc;
using System.Web.Routing;
namespace ControllersAndActions.Controllers
{
    public class BasicController : IController
    {
        public void Execute(RequestContext requestContext)
        {
            string controller = (string)requestContext.RouteData.Values["controller"];
            string action = (string)requestContext.RouteData.Values["action"];
            if (action.ToLower() == "redirect")
            {
                requestContext.HttpContext.Response.Redirect("/Derived/Index");
            }
            else
            {
                requestContext.HttpContext.Response.Write(
                    string.Format("Controller: {0}, Action: {1}",
                    controller, action));
            }
        }
    }
}
```

Вы можете использовать тот же подход, если вы унаследовали ваш контроллер от класса `Controller`. Класс `HttpResponseBase`, который возвращается, когда вы читаете свойство `requestContext.HttpContext.Response` в методе `Execute`, доступен через свойство `Controller.Response`, как показано в листинге 15-8, который демонстрирует улучшения в классе `DerivedController`.

**Листинг 15-8:** Генерирование выходных данных

```
using System.Web.Mvc;
```

```

namespace ControllersAndActions.Controllers
{
    public class DerivedController : Controller
    {
        public ActionResult Index()
        {
            ViewBag.Message = "Hello from the DerivedController Index method";
            return View("MyView");
        }
        public void ProduceOutput()
        {
            if (Server.MachineName == "TINY")
            {
                Response.Redirect("/Basic/Index");
            }
            else
            {
                Response.Write("Controller: Derived, Action: ProduceOutput");
            }
        }
    }
}

```

Метод `ProduceOutput` использует значение свойства `Server.MachineName` для определения того, какой ответ отправить клиенту. (`TINY` – это имя одного из наших компьютеров).

Такой подход работает, но тут возникает несколько проблем:

- Классы контроллеров должны содержать информацию об HTML или URL структуре, и поэтому эти классы труднее читать и поддерживать.
- Трудно провести модульное тестирование контроллера, который генерирует свой ответ непосредственно в выходные данные. Вам нужно создать mock-реализацию объекта `Response`, а затем быть в состоянии обработать выходные данные, полученные от контроллера, для того, чтобы определить, что представляет собой результат. Это может означать, например, разбор HTML по ключевым словам, что является длительным и болезненным процессом.
- Работа с мелкими деталями каждого ответа подобным образом является утомительной и может привести ко многим ошибкам. Некоторым программистам нравится абсолютный контроль, который дает создание «сырых» контроллеров, но нормальные люди довольно быстро разочаровываются в этом.

К счастью, MVC фреймворк предоставляет хорошую функциональную возможность, которая способствует решению всех этих вопросов, и она называется результаты действий. В следующих разделах введено понятие результата действия и показаны различные способы, которыми может быть использован результат действия для получения ответов от контроллеров.

## Что такое результаты действия

MVC фреймворк использует результаты действия, чтобы разделить заявление о намерениях от выполнения этих намерений. Она работает очень просто.

Вместо того чтобы работать непосредственно с объектом `Response`, мы возвращаем объект, наследованный от класса `ActionResult`, который описывает то, какой ответ от контроллера мы хотели бы увидеть, например, отображение представления или перенаправление на другой URL или метод действия.

## Примечание

Система результатов действия является примером *шаблона "Команда"* (Command pattern). Этот паттерн описывает сценарии, когда вы храните и передаете объекты, описывающие операции, которые должны быть выполнены. См. [http://en.wikipedia.org/wiki/Command\\_pattern](http://en.wikipedia.org/wiki/Command_pattern) для более подробной информации.

Когда MVC получает от метода действия объект ActionResult, он вызывает метод ExecuteResult, определяемый этим объектом. Реализация результата действия затем работает для вас с объектом Response, создавая выходные данные, которые соответствуют вашим намерениям. Простым примером является класс CustomRedirectResult, показанный в листинге 15-9, который мы определили в новой папке Infrastructure нашего проекта.

#### Листинг 15-9: Класс CustomRedirectResult

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
namespace ControllersAndActions.Infrastructure
{
    public class CustomRedirectResult : ActionResult
    {
        public string Url { get; set; }
        public override void ExecuteResult(ControllerContext context)
        {
            string fullUrl = UrlHelper.GenerateContentUrl(Url, context.HttpContext);
            context.HttpContext.Response.Redirect(fullUrl);
        }
    }
}
```

Мы взяли за основу работы этого класса способ, которым работает класс System.Web.Mvc.RedirectResult: одно из преимуществ того, что MVC является платформой с открытым исходным кодом, заключается в том, что вы можете сами посмотреть, как все это работает. Наш класс CustomRedirectResult намного проще, чем его MVC эквивалент, но этого достаточно для наших целей на данный момент.

Когда мы создаем экземпляр класса RedirectResult, мы передаем ему URL, на который мы хотим перенаправить пользователя. Метод ExecuteResult, который будет выполняться MVC фреймворком, когда наш метод действия закончит работу, получает объект Response для запроса через объект ControllerContext, который предоставляет фреймворк, и вызовет либо метод RedirectPermanent, либо Redirect, а это именно то, что мы делали вручную в листинге 15-8.

Вы можете увидеть, как мы использовали класс CustomRedirectResult, в листинге 15-10, который показывает, как мы изменили контроллер Derived.

#### Листинг 15-10: Использование класса CustomRedirectResult в контроллере Derived

```
using ControllersAndActions.Infrastructure;
using System.Web.Mvc;
namespace ControllersAndActions.Controllers
{
    public class DerivedController : Controller
    {
        public ActionResult Index()
        {
            ViewBag.Message = "Hello from the DerivedController Index method";
            return View("MyView");
        }
}
```

```

public ActionResult ProduceOutput()
{
    if (Server.MachineName == "TINY")
    {
        return new CustomRedirectResult { Url = "/Basic/Index" };
    }
    else
    {
        Response.Write("Controller: Derived, Action: ProduceOutput");
        return null;
    }
}
}
}

```

Заметьте, что нам нужно было изменить результат действия метода, чтобы вернуть `ActionResult`. Мы возвращаем `null`, если мы не хотим, чтобы MVC что-то делал, когда наш метод действия будет выполнен. Это то же, что мы и делаем, когда не возвращаем экземпляр `CustomRedirectResult`.

## Юнит тестирование контроллеров и действий

Многие части MVC фреймворка призваны облегчить модульное тестирование, и это особенно верно для действий и контроллеров. Есть несколько причин для этой поддержки:

- Вы можете проверить действия и контроллеры вне веб-сервера. Контекстные объекты доступны через свои базовые классы (например, `httpRequestBase`), для которых легко применима `mock`-технология.
- Вам не нужно разбирать какой-либо HTML для проверки результата метода действия. Вы можете проверить возвращаемый объект `ActionResult`, чтобы убедиться, что вы получили ожидаемый результат.
- Вам не нужно имитировать запросы клиентов. Система связывания данных модели MVC фреймворка позволяет вам писать методы действий, которые получают входные данные в качестве параметров метода. Для проверки метода действия вы просто вызываете метод действия и передаете значения параметров, которые вас интересуют.

Мы покажем вам, как создавать юнит тесты для различных видов результатов по мере продвижения по данной главе.

Не забывайте, что модульное тестирование – это не полная история. Сложные виды поведения в приложении возникают тогда, когда методы действий вызываются последовательно. Модульное тестирование лучше всего сочетается с другими видами тестирования.

Теперь, когда вы увидели, как работает наш пользовательский результат действия, мы можем переключиться на один из предоставляемых MVC, который обладает расширенным набором функций и был тщательно протестирован Microsoft. Листинг 15-11 показывает изменения, которые мы внесли.

### Листинг 15-11: Использование встроенного объекта `RedirectResult`

```

...
public ActionResult ProduceOutput() {
    return new RedirectResult("/Basic/Index");
}
...

```

Мы удалили условный оператор из метода действия, а это значит, что если вы запустите приложение и перейдите к методу /Derived/ProduceOutput, ваш браузер будет перенаправлен на URL /Basic/Index.

Чтобы упростить код метода действия, класс Controller включает в себя методы для создания различных видов объектов ActionResult. Так, в качестве примера, мы можем добиться результата из листинга 15-11, вернув результат метода Redirect, как показано в листинге 15-12.

**Листинг 15-12:** Использование метода контроллера для создания результата действия

```
...
public ActionResult ProduceOutput() {
    return Redirect("/Basic/Index");
}
...
```

Нет ничего особенно сложного в системе результатов действий, но это поможет вам в конечном итоге получить более простой, более чистый и более последовательный код, который легче читать и который проще для модульного тестирования.

В случае переадресации, например, вы можете просто проверить, что метод действия возвращает экземпляр RedirectResult и что свойство Url содержит ожидаемую цель.

MVC фреймворк содержит ряд встроенных типов результатов действий, которые показаны в таблице 15-2. Все эти типы являются производными от ActionResult, и многие из них имеют удобные вспомогательные методы в классе Controller. В следующих разделах мы покажем вам, как использовать эти типы результатов.

**Таблица 15-2:** Встроенные типы ActionResult

Тип	Описание	Вспомогательные методы
ViewResult	Отображает указанный шаблон представления или шаблон по умолчанию	View
PartialViewResult	Отображает указанный шаблон частичного представления или шаблон PartialView по умолчанию	
RedirectToRouteResult	Работает с HTTP перенаправлением 301 или 302 на метод действия или конкретный роут, генерируя URL в соответствии с вашей конфигурацией	RedirectToAction, RedirectToActionPermanent, RedirectToRoute, RedirectToRoutePermanent
RedirectResult	Работает с HTTP перенаправлением 301 или 302 на конкретный URL	Redirect, RedirectPermanent
HttpUnauthorizedResult	Устанавливает ответный код HTTP статуса на 401 (что означает "не авторизован"), который вызывает активный механизм аутентификации (form-аутентификацию или Windows-аутентификацию), чтобы попросить посетителя войти в систему	Нет
HttpNotFoundResult	Возвращает HTTP ошибку 404-Not found	HttpNotFound
HttpStatusCodeResult	Возвращает указанный HTTP код	Нет

Тип	Описание	Вспомогательные методы
EmptyResult	Ничего не делает	Нет

## Возвращение HTML при отображении представления

Наиболее распространенный вид ответа от метода действия заключается в создании HTML и отправке его браузеру. При использовании системы результатов действий вы делаете это путем создания экземпляра класса `ViewResult`, который определяет желаемое представление для генерирования HTML, как показано в листинге 15-13.

### Листинг 15-13: Указание отображаемого представления при помощи `ViewResult`

```
using System.Web.Mvc;
namespace ControllersAndActions.Controllers
{
    public class ExampleController : Controller
    {
        public ViewResult Index()
        {
            return View("Homepage");
        }
    }
}
```

В этом листинге мы используем вспомогательный метод `View` для создания экземпляра класса `ViewResult`, который затем возвращается как результат действия метода.

### Примечание

Обратите внимание, что возвращаемым типом метода действия в листинге является `ViewResult`. Метод компилировался бы и работал так же хорошо, если бы мы указали более общий тип `ActionResult`. На самом деле некоторые MVC программисты определяют результат каждого метода действия как `ActionResult`, даже если они знают, что всегда будет возвращаться более конкретный тип. В следующих примерах мы все же будем более усердны, чтобы показать вам, как можно использовать каждый тип результата, но мы, как правило, более спокойно относимся к этому в реальных проектах.

Вы указываете представление, которое нужно отобразить, с помощью параметра метода `View`. В этом примере мы указали представление `Homepage`.

### Примечание

Мы могли бы создать объект `ViewResult` явно, при помощи `return new ViewResult { ViewName = "Homepage" };`. Это вполне приемлемый подход, но мы предпочитаем использовать вспомогательные методы, определенные в классе `Controller`.

Когда MVC вызывает метод `ExecuteResult` объекта `ViewResult`, начинается поиск представления, которое вы указали. Если вы используете в вашем проекте области, то фреймворк будет искать в следующих местах:

- /Areas/<AreaName>/Views/<ControllerName>/<ViewName>.aspx
- /Areas/<AreaName>/Views/<ControllerName>/<ViewName>.ascx
- /Areas/<AreaName>/Views/Shared/<ViewName>.aspx
- /Areas/<AreaName>/Views/Shared/<ViewName>.ascx
- /Areas/<AreaName>/Views/<ControllerName>/<ViewName>.cshtml

- /Areas/<AreaName>/Views/<ControllerName>/<ViewName>.vbhtml
- /Areas/<AreaName>/Views/Shared/<ViewName>.cshtml
- /Areas/<AreaName>/Views/Shared/<ViewName>.vbhtml

Вы можете видеть из списка, что фреймворк ищет представления, которые были созданы для движка представления ASPX (расширения файлов .aspx и .ascx), даже если был указан Razor, когда создавался проект. Также фреймворк ищет C# и Visual Basic .NET шаблоны Razor (.cshtml файлы – это C#, а .vbhtml – Visual Basic; синтаксис Razor такой же в этих файлах, но фрагменты кода, как можно понять из названий, написаны на разных языках). MVC проверяет по очереди, существует ли каждый из этих файлов. Как только он находит соответствие, он использует представление, чтобы отобразить результат метода действия.

Если вы не используете области или вы используете области, но ни один из файлов в предыдущем списке не был найден, тогда фреймворк продолжит свой поиск в следующих местах:

- /Views/<ControllerName>/<ViewName>.aspx
- /Views/<ControllerName>/<ViewName>.ascx
- /Views/Shared/<ViewName>.aspx
- /Views/Shared/<ViewName>.ascx
- /Views/<ControllerName>/<ViewName>.cshtml
- /Views/<ControllerName>/<ViewName>.vbhtml
- /Views/Shared/<ViewName>.cshtml
- /Views/Shared/<ViewName>.vbhtml

Еще раз, как только MVC фреймворк проверит расположение файлов и найдет нужный файл, то поиск прекратится, и найденное представление будет использоваться для отображения ответа клиенту.

Мы не используем в нашем примере приложения области, поэтому фреймворк будет сразу искать в /Views/Example/Index.aspx. Обратите внимание, что опущена часть Controller имени класса, поэтому создание ViewResult в ExampleController приводит к поиску каталога с именем Example.

## Юнит тест: отображение представления

Для проверки представления, которое отображает метод действия, вы можете проверить объект ViewResult, который он возвращает. Возможно, это не совсем тот тест, который вы ожидали увидеть, ведь вы не проходите по всему процессу до проверки сгенерированного HTML, но он достаточно хорош, пока у вас есть уверенность в том, что MVC фреймворк работает должным образом. Мы добавили новый проект Unit Test в Visual Studio и добавили файл ActionTests.cs, чтобы в нем содержались наши методы тестирования.

Первая ситуация для проверки появляется тогда, когда метод действия выбирает конкретное представление:

```
public ViewResult Index() {
    return View("Homepage");
}
```

Вы можете определить, какое представление было выбрано, прочитав свойство ViewName объекта ViewResult, как показано в данном тестовом методе.

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using ControllersAndActions.Controllers;
using System.Web.Mvc;
```

```

namespace ControllersAndActions.Tests
{
    [TestClass]
    public class ActionTests
    {
        [TestMethod]
        public void ViewSelectionTest()
        {
            // Arrange - создание контроллера
            ExampleController target = new ExampleController();
            // Act - вызов метода действия
            ViewResult result = target.Index();
            // Assert - проверка результата
            Assert.AreEqual("Homepage", result.ViewName);
        }
    }
}

```

Небольшое различие возникает тогда, когда вы тестируете метод действия, который выбирает представление по умолчанию:

```

public ViewResult Index() {
    return View();
}

```

В такой ситуации вы должны принять пустую строку ("") для названия представления:

```
Assert.AreEqual("", result.ViewName);
```

Последовательность директорий, в которой MVC ищет представление, является еще одним примером соглашения о конфигурации. Вам не нужно регистрировать файлы представлений при помощи фреймворка. Вы просто добавляете их в одно из множества известных мест, и фреймворк их найдет. Мы можем пройти на шаг вперед в этом соглашении, опуская имя представления, которое мы хотим отобразить, когда мы вызываем метод `View`, как показано в листинге 15-14.

#### **Листинг 15-14:** Создание `ViewResult` без указания представления

```

using System.Web.Mvc;
using System;
namespace ControllersAndActions.Controllers
{
    public class ExampleController : Controller
    {
        public ViewResult Index()
        {
            return View();
        }
    }
}

```

Когда мы делаем так, MVC предполагает, что мы хотим отобразить представление, которое имеет такое же имя, что и метод действия. Это означает, что вызов метода `View` в листинге 15-14 запускает поиск представления `Index`.

#### **Примечание**

*Результат этого заключается в том, что ищется представление, которое имеет такое же имя, что и метод действия, но имя представления фактически*

*определяется из значения `RouteData.Values["action"]`, которое мы объяснили как часть системы маршрутизации в главах 13 и 14.*

---

Есть несколько перегруженных версий метода `View`. Они соответствуют настройкам различных свойств объекта `ViewResult`, который создается. Например, вы можете переопределить макет, который используется представлением, явно указав альтернативу:

```
...
public ViewResult Index() {
    return View("Index", "_AlternateLayoutPage");
}
...
```

### Указание представления по его пути

Подход с использованием соглашения по именованию удобен и прост, но таким образом ограничивается число представлений, которые вы можете отобразить. Если вы хотите отобразить конкретное представление, вы можете сделать это, если явно укажете путь к нему и обойдете фазу поиска. Вот пример:

```
using System.Web.Mvc;
namespace ControllersAndActions.Controllers
{
    public class ExampleController : Controller
    {
        public ViewResult Index()
        {
            return View("~/Views/Other/Index.cshtml");
        }
    }
}
```

При указании представления наподобие этого путь должен начинаться с `/` или `~/` и включать в себя расширение имени файла (например, `.cshtml` для представлений Razor, содержащих C# код).

Если вы собираетесь использовать эту функцию, мы предлагаем вам сделать маленькую паузу и спросить себя, чего вы пытаетесь достичь. Если вы пытаетесь отобразить представление, которое принадлежит другому контроллеру, то лучше перенаправить пользователя на метод действия в этом контроллере (см. раздел "Перенаправление на метод действия" далее в этой главе). Если вы пытаетесь обойти схему именований, потому что данный способ не подходит тому, как вы организовали ваш проект, то см. главу 18, в которой объясняется, как реализовать пользовательскую последовательность поиска.

### Передача данных из метода действия в представление

Нам часто нужно передать данные из метода действия в представление. MVC Framework предлагает несколько различных способов сделать это, о чем мы и поговорим в следующих разделах. В этих разделах мы коснемся темы представлений, которые мы подробно рассмотрим в главе 18. В этой главе мы затронем только тот функционал представлений, который необходим для демонстрации интересующих нас возможностей контроллеров.

#### Предоставление объекта модели представления

Вы можете отправить объект в представление, передав его в качестве параметра методу `View`, как показано в листинге 15-15.

### Листинг 15-15: Указание объекта модели представления

```
...
public ViewResult Index() {
    DateTime date = DateTime.Now;
    return View(date);
}
...
```

Мы передали объект `DateTime` в качестве модели представления. Мы можем обратиться к объекту в представлении, используя ключевое слово `Razor Model`, как показано в листинге 15-16.

### Листинг 15-16: Доступ к модели представления в Razor представлении

```
@{
    ViewBag.Title = "Index";
}
<h2>Index</h2>
The day is: @((DateTime)Model).DayOfWeek
```

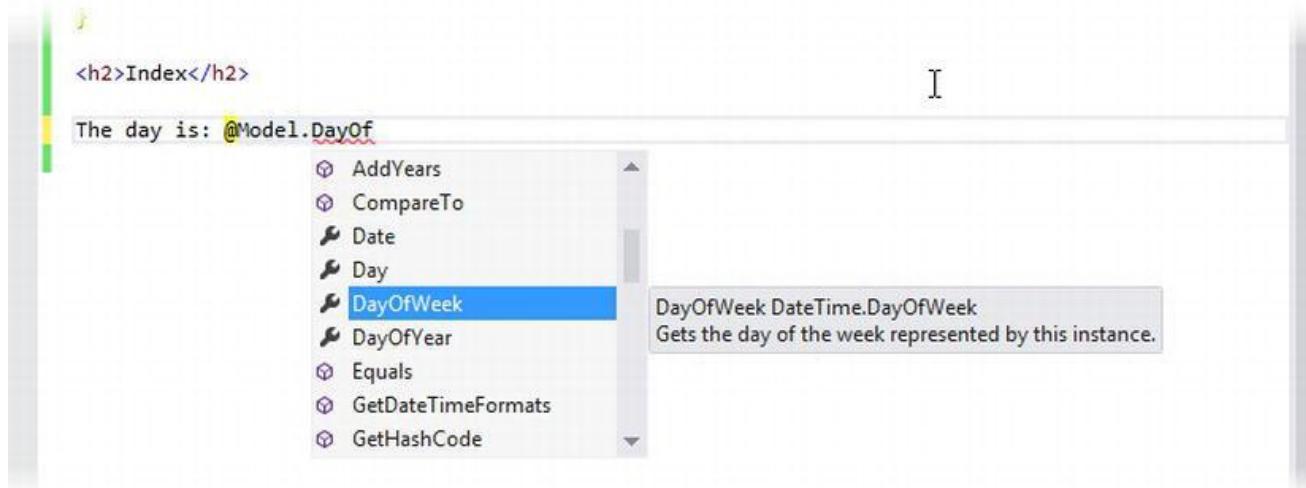
Представление, показанное в листинге 15-16, известно как *нетипизированное или слабо типизированное*. Представление ничего не знает об объекте модели представления и обрабатывает его как экземпляр `object`. Чтобы получить значение свойства `DayOfWeek`, нам нужно привести объект к экземпляру `DateTime`. Это работает, но представление получается «грязным». Мы можем изменить это, создавая строго типизированные представления, когда мы говорим представлениям, каким будет тип объекта модели представления, как показано в листинге 15-17.

### Листинг 15-17: Строго типизированное представление

```
@model DateTime
 @{
    ViewBag.Title = "Index";
}
<h2>Index</h2>
The day is: @Model.DayOfWeek
```

Мы указываем тип модели представления с помощью ключевого слова `Razor model`. Обратите внимание, что мы используем строчную `m`, когда указываем тип модели, и заглавную `M`, когда мы читаем значение. Не только это помогает очистить наше представление, также Visual Studio поддерживает IntelliSense для строго типизированных представлений, как показано на рисунке 15-3.

### Рисунок 15-3: Поддержка IntelliSense для строго типизированных представлений



## Юнит тест: объекты модели представления

Вы можете получить доступ к объекту модели представления, переданного методом действия представлению, при помощи свойства `ViewResult.ViewData.Model`. Вот простой метод действия:

```
public ViewResult Index() {
    return View((object)"Hello, World");
}
```

Этот метод действия передает `string` в качестве объекта модели представления. Мы привели его к `object`, так что компилятор не думает, что мы хотим перегруженный вариант `View`, который указывает имя представления. Мы можем получить доступ к объекту модели представления с помощью свойства `ViewData.Model`, как показано в данном тестовом методе:

```
...
[TestMethod]
public void ViewSelectionTest() {
    // Arrange - создание контроллера
    ExampleController target = new ExampleController();
    // Act - вызов метода действия
    ViewResult result = target.Index();
    // Assert - проверка результата
    Assert.AreEqual("Hello, World", result.ViewData.Model);
}
...
```

## Передача данных при помощи ViewBag

Мы представили `ViewBag` в главе 2. Эта возможность позволяет определять произвольные свойства динамического объекта и дает доступ к ним в представлении. Доступ к динамическому объекту можно получить через свойство `Controller.ViewBag`, как показано в листинге 15-18.

### Листинг 15-18: Использование ViewBag

```
using System;
using System.Web.Mvc;
namespace ControllersAndActions.Controllers
{
    public class ExampleController : Controller
    {
        public ViewResult Index()
        {
            ViewBag.Message = "Hello";
            ViewBag.Date = DateTime.Now;
            return View();
        }
    }
}
```

В листинге мы определили свойства `Message` и `Date`, просто присвоив им значения. До этого момента таких свойств не существовало, и мы не делали никаких приготовлений для их создания. Чтобы прочитать данные обратно в представлении, мы просто получаем те же свойства, которые мы установили в методе действия, как показано в листинге 15-19.

### Листинг 15-19: Чтение данных из ViewBag

```
@{
    ViewBag.Title = "Index";
}
<h2>Index</h2>
```

```
The day is: @ViewBag.Date.DayOfWeek  
<p />  
The message is: @ViewBag.Message
```

ViewBag имеет преимущество перед использованием объекта модели представления в том, что таким образом легко отправлять несколько объектов в представление. Если бы мы были ограничены в использовании моделей представления, то мы должны были бы создать новый тип, который имеет членов `string` и `DateTime`, чтобы получить тот же результат, что и в листингах 15-18 и 15-19.

При работе с динамическими объектами вы можете ввести в представление любую последовательность вызовов методов и свойств:

```
The day is: @ViewBag.Date.DayOfWeek.Blah.Blah.Blah
```

Visual Studio не обеспечивает поддержку IntelliSense для любых динамических объектов, в том числе `ViewBag`, и такие ошибки, как эта, не будут показаны, пока представление не будет отображено.

Нам нравится гибкость `ViewBag`, но мы склонны придерживаться строго типизированных представлений. Нет никаких ограничений, которые могли бы остановить нас от использования и моделей представления, и `ViewBag` в том же представлении. Обе эти возможности могут реализованы рядом друг с другом без проблем.

## Юнит тест: ViewBag

Вы можете прочитать значения из `ViewBag` через свойство `ViewResult.ViewBag`. Следующий тестовый метод предназначен для метода действия из листинга 15-18:

```
[TestMethod]  
public void ViewSelectionTest() {  
    // Arrange - создание контроллера  
    ExampleController target = new ExampleController();  
    // Act - вызов метода действия  
    ActionResult result = target.Index();  
    // Assert - проверка результата  
    Assert.AreEqual("Hello", result.ViewBag.Message);  
}  
...
```

## Выполнение перенаправлений

Частый результат метода действия заключается не в том, чтобы произвести какие-либо выходные данные напрямую, а в том, чтобы перенаправить браузер пользователя на другой URL. В большинстве случаев этот URL является еще одним методом действие в приложении, который генерирует выходные данные, которые должны быть показаны пользователю.

## Паттерн POST/REDIRECT/GET

Наиболее часто перенаправление используется в методах действия, которые обрабатывают запросы HTTP `POST`. Как мы упоминали в предыдущей главе, запросы `POST` используются тогда, когда вы хотите изменить состояние приложения. Если вы просто возвращаете HTML после обработки запроса, вы рискуете, что пользователь будет нажимать кнопку перезагрузки браузера и повторит отправку форму во второй раз, вызывая неожиданные и нежелательные результаты.

Чтобы избежать этой проблемы, вы можете следовать паттерну *Post/Redirect/Get*. В этом паттерне вы получаете `POST` запрос, обрабатываете его, а затем перенаправляете браузер так, чтобы `GET` запрос был

сделан браузером для другого URL. GET запросы не должны изменять состояние приложения, так что любая случайная повторная отправка этого запроса не вызовет никаких проблем.

Когда вы выполняете перенаправление, вы отправляете браузеру один из двух HTTP кодов:

- HTTP код 302, который является *временным* перенаправлением. Это наиболее часто используемый тип перенаправления и при использовании паттерна Post/Redirect/Get, это именно тот код, который вы хотите отправить.
- HTTP код 301, который указывает на *постоянное* перенаправление. Его следует использовать с осторожностью, поскольку он говорит получателю HTTP кода никогда снова не запрашивать оригинальный URL, а использовать новый URL, который включен в код перенаправления. Если вы сомневаетесь, используйте временное перенаправление, то есть, отправляйте код 302.

## Перенаправление на URL литерального формата

Самый простой способ перенаправить браузер заключается в вызове метод `Redirect`, который возвращает экземпляр класса `RedirectResult`, как показано в листинге 15-20.

### Листинг 15-20: Перенаправление на URL литерального формата

```
using System;
using System.Web.Mvc;
namespace ControllersAndActions.Controllers
{
    public class ExampleController : Controller
    {
        public ViewResult Index()
        {
            ViewBag.Message = "Hello";
            ViewBag.Date = DateTime.Now;
            return View();
        }
        public RedirectResult Redirect()
        {
            return Redirect("/Example/Index");
        }
    }
}
```

URL, на который вы хотите сделать перенаправление, выражается в виде строки и передается в качестве параметра методу `Redirect`. Метод `Redirect` реализует временное перенаправление. Вы можете создать постоянное перенаправление, используя метод `RedirectPermanent`, как показано в листинге 15-21.

### Листинг 15-21: Реализация постоянного перенаправления на литеральный URL

```
...
public RedirectResult Redirect() {
    return RedirectPermanent("/Example/Index");
}
...
```

#### *Совет*

*Если вы хотите, вы можете использовать перегруженную версию метода `Redirect`. Она принимает параметр `bool`, который определяет, является ли перенаправление постоянным или нет.*

## Юнит тест: перенаправление на URL литерального формата

Перенаправление на URL литерального формата легко протестировать. Вы можете прочитать URL и проверить, является ли перенаправление постоянным или времененным, используя свойства `Url` и `Permanent` класса `RedirectResult`. Ниже приведен тестовый метод для перенаправления, показанного в листинге 15-21:

```
...
[TestMethod]
public void RedirectTest() {
    // Arrange - create the controller
    ExampleController target = new ExampleController();
    // Act - call the action method
    RedirectResult result = target.Redirect();
    // Assert - check the result
    Assert.IsFalse(result.Permanent);
    Assert.AreEqual("/Example/Index", result.Url);
}
...
```

## Перенаправление на URL системы маршрутизации

Если вы перенаправляете пользователя в другую часть вашего приложения, вы должны убедиться, что URL, который вы отправляете, действителен внутри вашей URL схемы, как было описано в предыдущей главе. Проблема с использованием URL литерального формата для перенаправления заключается в том, что любые изменения в своей схеме маршрутизации приводят к тому, что вы должны пройти по всему коду и обновить URL.

В качестве альтернативы вы можете использовать систему маршрутизации, чтобы генерировать правильные URL при помощи метода `RedirectToRoute`, который создает экземпляр `RedirectToRouteResult`, как показано в листинге 15-22.

### Листинг 15-22: Перенаправление на URL системы маршрутизации

```
using System;
using System.Web.Mvc;
namespace ControllersAndActions.Controllers
{
    public class ExampleController : Controller
    {
        public ViewResult Index()
        {
            ViewBag.Message = "Hello";
            ViewBag.Date = DateTime.Now;
            return View();
        }
        public RedirectToRouteResult Redirect()
        {
            return RedirectToAction(new
            {
                controller = "Example",
                action = "Index",
                ID = "MyID"
            });
        }
    }
}
```

Метод `RedirectToRoute` работает с временным перенаправлением. Используйте метод `RedirectToRoutePermanent` для постоянного перенаправления. Оба метода принимают анонимный

типа, чьи свойства затем передаются системе маршрутизации для генерации URL. Более подробную информацию об этом процессе см. в главе 14.

## Юнит тестирование: роутовые перенаправления

Вот юнит тест, который мы добавили в файл `ActionTests.cs`. Он тестирует метод действия из листинга 15-22:

```
...
[TestMethod]
public void RedirectToValueTest() {
    // Arrange - create the controller
    ExampleController target = new ExampleController();
    // Act - call the action method
    RedirectToRouteResult result = target.RedirectToRoute();
    // Assert - check the result
    Assert.IsFalse(result.Permanent);
    Assert.AreEqual("Example", result.RouteValues["controller"]);
    Assert.AreEqual("Index", result.RouteValues["action"]);
    Assert.AreEqual("MyID", result.RouteValues["ID"]);
}
...
...
```

## Перенаправление на метод действия

Вы можете сделать перенаправление на метод действия более изящно, используя метод `RedirectToAction`. Это всего лишь обертка вокруг метода `RedirectToRoute`, который позволяет указать значения для метода действия и контроллера без необходимости создания анонимного типа, как показано в листинге 15-23.

### Листинг 15-23: Перенаправление при помощи `RedirectToAction`

```
...
public RedirectToRouteResult RedirectToRoute() {
    return RedirectToAction("Index");
}
...
```

Если вы просто указываете метод действия, то предполагается, что вы имеете в виду метод действия в текущем контроллере. Если вы хотите сделать перенаправление на другой контроллер, необходимо указать имя в качестве параметра:

```
...
public RedirectToRouteResult Redirect() {
    return RedirectToAction("Index", "Basic");
}
...
```

Есть и другие перегруженные версии, которые можно использовать для добавления дополнительных значений при создании URL. Они выражаются при помощи анонимного типа, который имеет тенденцию скрывать цели метода, но все же может сделать ваш код более удобным для чтения.

### Примечание

*Значения, которые вы передаете методу действия и контроллеру, не проверяются, прежде чем передаются системе маршрутизации. Вы несете ответственность за то, чтобы цели, которые вы определяете, действительно существовали.*

Метод `RedirectToAction` выполняет временное перенаправление. Для постоянного перенаправления используйте `RedirectToActionPermanent`.

## Сохранение данных во время перенаправления

Перенаправление заставляет браузер отправить совершенно новый HTTP запрос, а это означает, что у вас нет доступа к данным оригинального запроса. Если вы хотите передать данные одного запроса другому, вы можете использовать `TempData`.

Данные `TempData` похожи на данные `Session`, за исключением того, что значения `TempData` помечаются на удаление, после того как они прочитаны, и они удаляются, когда запрос был обработан. Это идеальное средство для сохранения краткосрочных данных, которые необходимо оставить во время выполнения перенаправления. Вот простой пример в методе действия, который использует метод `RedirectToAction`:

```
...
public RedirectToRouteResult RedirectToRoute() {
    TempData["Message"] = "Hello";
    TempData["Date"] = DateTime.Now;
    return RedirectToAction("Index");
}
...
```

Когда этот метод обрабатывает запрос, он устанавливает значения в коллекции `TempData`, а затем перенаправляет браузер пользователя на метод действия `Index` в том же контроллере. Вы можете прочитать значения `TempData` обратно в целевом методе действия, а затем передать их в представление:

```
...
public ViewResult Index() {
    ViewBag.Message = TempData["Message"];
    ViewBag.Date = TempData["Date"];
    return View();
}
...
```

Более прямым подходом является прочтение этих значений в представлении:

```
@{
    ViewBag.Title = "Index";
}
<h2>Index</h2>
The day is: @((DateTime)TempData["Date"]).DayOfWeek
<p />
The message is: @TempData["Message"]
```

Прочтение этих значений в представлении означает, что вам не нужно использовать `ViewBag` в методе действия. Тем не менее, необходимо привести результаты `TempData` к соответствующему типу.

Вы можете получить значения из `TempData` без маркировки их на удаление с помощью метода `Peek`:

```
DateTime time = (DateTime)TempData.Peek("Date");
```

Можно сохранить значения, которое в противном случае были бы удалены, с помощью метода `Keep`:

```
TempData.Keep("Date");
```

Метод `Keep` не защищает значение навсегда. Если значение будет прочтено еще раз, оно будет помечено для удаления еще раз. Если вы хотите сохранить данные, чтобы они не были удалены после обработки запроса, используйте данные `Session`.

## Возвращение ошибок и HTTP кодов

Последний из встроенных классов `ActionResult`, которые мы рассмотрим, может быть использован для отправки клиенту сообщений об ошибках и результирующих HTTP кодов. Большинство приложений не требуют этой функции, потому что MVC автоматически генерирует такие виды результатов. Однако они могут быть полезны, если вам нужен более прямой контроль над ответами, которые отправляются клиенту.

### Отправка конкретного результирующего HTTP кода

Вы можете отправить конкретный код HTTP статуса браузеру с помощью класса `HttpStatusCodeResult`. Для этого не существует вспомогательного метода контроллера, так что вы должны создать экземпляр класса напрямую, как показано в листинге 15-24.

#### Листинг 15-24: Отправка конкретного кода статуса

```
using System;
using System.Web.Mvc;
namespace ControllersAndActions.Controllers
{
    public class ExampleController : Controller
    {
        public ViewResult Index()
        {
            ViewBag.Message = "Hello";
            ViewBag.Date = DateTime.Now;
            return View();
        }
        public RedirectToRouteResult Redirect()
        {
            return RedirectToAction("Index", "Basic");
        }
        public HttpStatusCodeResult StatusCode()
        {
            return new HttpStatusCodeResult(404, "URL cannot be serviced");
        }
    }
}
```

Параметры конструктора для `HttpStatusCodeResult` являются числовым кодом статуса и дополнительным описательным сообщением. В листинге мы вернули код 404, что означает, что запрашиваемый ресурс не существует.

### Отправка результата 404

Мы можем достичь того же результата, что показан в листинге 15-24, с помощью более удобного класса `HttpNotFoundResult`, который является производным от `HttpStatusCodeResult` и может быть создан с помощью метода контроллера `NotFound`, как показано в листинге 15-25.

#### Листинг 15-25: Создание результата 404

```
...
public HttpStatusCodeResult StatusCode()
{
    return HttpNotFound();
}
```

...

## Отправка результата 401

Другим классом-«оберткой» для конкретного кода HTTP статуса является `HttpUnauthorizedResult`, который возвращает код 401, используемый для указания того, что запрос является неавторизованным. В листинге 15-26 показан пример.

### Листинг 15-26: Генерирование результата 401

```
...
public HttpStatusCodeResult StatusCode() {
    return new HttpUnauthorizedResult();
}
...
```

В классе `Controller` нет вспомогательных методов для создания экземпляров `HttpUnauthorizedResult`, так что вы должны делать это напрямую. Результат возвращения экземпляра этого класса, как правило, заключается в перенаправлении пользователя на страницу аутентификации, как вы видели в главе 11.

## Юнит тест: коды HTTP статуса

Класс `HttpStatusCodeResult` следует тому паттерну, который используется и для других типов результата, и его статус доступен через набор свойств. В данном случае свойство `StatusCode` возвращает числовой код HTTP статуса, а свойство `StatusDescription` возвращает соответствующее описание. Следующий тестовый метод предназначен для метода действия из листинга 15-26:

```
...
[TestMethod]
public void StatusCodeResultTest() {
    // Arrange - create the controller
    ExampleController target = new ExampleController();
    // Act - call the action method
    HttpStatusCodeResult result = target.StatusCode();
    // Assert - check the result
    Assert.AreEqual(401, result.StatusCode);
}
```

## Резюме

Контроллеры являются одним из ключевых элементов MVC паттерна. В этой главе вы узнали, как создавать "сырые" контроллеры путем реализации интерфейса `IController` и более удобные контроллеры путем наследования от класса `Controller`. Вы увидели ту роль, которую играют методы действия в контроллерах MVC фреймворка, и то, как они облегчают модульное тестирование. Мы показали вам различные способы, которыми вы можете получать входные данные и формировать выходные данные от методов действий, и продемонстрировали различные виды `ActionResult`, которые делают этот процесс простым и гибким.

В следующей главе мы глубже рассмотрим контроллеры, чтобы показать вам *фильтры*. Они меняют порядок, в котором обрабатываются запросы.

# Фильтры

Фильтры внедряют дополнительную логику в процесс обработки запросов MVC Framework. Они обеспечивают простой и эффективный механизм для реализации *сквозной функциональности* (*cross-cutting concerns*). Под этим термином подразумевается функциональность, которая используется везде в приложении и в то же время не может быть заключена в какой-либо один из его модулей, так как это нарушит *принцип разделения ответственности* (*separation of concerns*). Классическими примерами сквозной функциональности являются вход в систему, авторизация и кэширование. В этой главе мы рассмотрим различные категории фильтров, которые поддерживает MVC Framework, научимся их создавать, использовать и контролировать их исполнение.

## Использование фильтров

Вы уже видели пример фильтра в главе 11, когда мы применили авторизацию к контроллеру администрирования SportsStore. Мы хотели, чтобы данный метод действия использовался только прошедшими аутентификацией пользователями, и это можно было бы реализовать несколькими способами. Например, можно проверять статус авторизованности запроса в каждом методе действия, как показано в листинге 16-1.

**Листинг 16-1:** Явная проверка авторизации в методах действия

```
namespace SportsStore.WebUI.Controllers
{
    public class AdminController : Controller
    {
        // ... instance variables and constructor
        public ViewResult Index()
        {
            if (!Request.IsAuthenticated)
            {
                FormsAuthentication.RedirectToLoginPage();
            }
            // ...rest of action method
        }
        public ViewResult Create()
        {
            if (!Request.IsAuthenticated)
            {
                FormsAuthentication.RedirectToLoginPage();
            }
            // ...rest of action method
        }
        public ViewResult Edit(int productId)
        {
            if (!Request.IsAuthenticated)
            {
                FormsAuthentication.RedirectToLoginPage();
            }
            // ...rest of action method
        }
        // ... other action methods
    }
}
```

Как видите, при таком подходе много повторов, и вместо этого разумнее использовать фильтр, как показано в листинге 16-2.

## Листинг 16-2: Применяем фильтр

```
namespace SportsStore.WebUI.Controllers
{
    [Authorize]
    public class AdminController : Controller
    {
        // ... instance variables and constructor
        public ViewResult Index()
        {
            // ...rest of action method
        }
        public ViewResult Create()
        {
            // ...rest of action method
        }
        public ViewResult Edit(int productId)
        {
            // ...rest of action method
        }
        // ... other action methods
    }
}
```

Фильтры – это атрибуты .NET, которые добавляют дополнительные этапы в конвейер обработки запроса. В листинге 16-2 мы использовали фильтр `Authorize`, который имеет тот же эффект, как и все дублирующиеся проверки в листинге 16-1.

## Памятка по атрибутам .NET

Атрибуты – это специальные классы .NET, наследующие от `System.Attribute`. Вы можете применить их к другим элементам кода, в том числе классам, методам, свойствам и полям. Их цель состоит в том, чтобы внедрить дополнительную информацию в скомпилированный код, которую можно позже прочитать в среде выполнения.

В C# для применения атрибутов используются квадратные скобки, а заполнить их общедоступные свойства можно путем присвоения значений параметрам (например, `[MyAttribute(SomeProperty=value)]`). Согласно соглашению по именам компилятора C#, если имя класса атрибута заканчивается на `Attribute`, эту часть можно опустить (например, применить фильтр `AuthorizeAttribute` можно, записав только `[Authorize]`).

## Четыре основных типа фильтров

MVC Framework поддерживает четыре типа фильтров, которые позволяют внедрять логику в разное время процесса обработки запроса. Они описаны в таблице 16-1.

Таблица 16-1: Типы фильтров MVC Framework

Тип фильтра	Интерфейс	Реализация по умолчанию	Описание
Фильтр авторизации	<code>IAuthorizationFilter</code>	<code>AuthorizeAttribute</code>	Запускается вначале, перед любым другим фильтром или методом действия
Фильтр действий	<code>IActionFilter</code>	<code>ActionFilterAttribute</code>	Запускается до и после метода действия
Фильтр результатов	<code>IResultFilter</code>	<code>ActionFilterAttribute</code>	Запускается до и после выполнения результата действия

Тип фильтра	Интерфейс	Реализация по умолчанию	Описание
Фильтр исключений	IExceptionFilter	HandleErrorAttribute	Запускается только в том случае, если другой фильтр, метод действия или результат действия генерирует исключение

Перед тем, как вызвать действие, MVC Framework проверяет определение метода на наличие атрибутов, реализующих перечисленные в таблице 16-1 интерфейсы. Если они есть, то в соответствующий момент обработки запроса вызывается метод, определенный этим интерфейсом. Платформа включает стандартные классы атрибутов, которые реализуют интерфейсы фильтров. Позже в данной главе мы научимся использовать эти классы.

#### Примечание

*Класс ActionFilterAttribute реализует и интерфейс IActionFilter, и IResultFilter. Этот класс является абстрактным, что подразумевает необходимость его реализовать. Другие классы с полезными функциями, такие как AuthorizeAttribute и HandleErrorAttribute, можно использовать, не создавая производный класс.*

## Применяем фильтры к контроллерам и методам действий

Фильтры можно применить к отдельным методам действий или к целому контроллеру. В листинге 16-2 мы применили фильтр Authorize к классу AdminController. Это имеет тот же эффект, как и применение его к каждому методу действия в контроллере, что показано в листинге 16-3.

#### Листинг 16-3: Применяем фильтр к методам действий индивидуально

```
namespace SportsStore.WebUI.Controllers
{
    public class AdminController : Controller
    {
        // ... instance variables and constructor
        [Authorize]
        public ViewResult Index()
        {
            // ...rest of action method
        }

        [Authorize]
        public ViewResult Create()
        {
            // ...rest of action method
        }
        // ... other action methods
    }
}
```

Вы можете одновременно применить несколько фильтров, причем на разных уровнях, то есть и к контроллеру, и отдельному методу действия. В листинге 16-4 показано использование трех различных фильтров.

#### Листинг 16-4: Применяем несколько фильтров в классе контроллера

```
[Authorize(Roles = "trader")] // applies to all actions
public class ExampleController : Controller
{
    [ShowMessage] // applies to just this action
    [OutputCache(Duration = 60)] // applies to just this action
    public ActionResult Index()
    {
        // ... action method body
    }
}
```

Некоторые фильтры из листинга 16-4 принимают параметры. Мы рассмотрим, как они работают, когда будем изучать виды фильтров по отдельности.

#### Примечание

*Если вы определили пользовательский базовый класс для контроллеров, фильтры, примененные к базовому классу, будут использоваться и в производных классах.*

## Создание проекта для примера

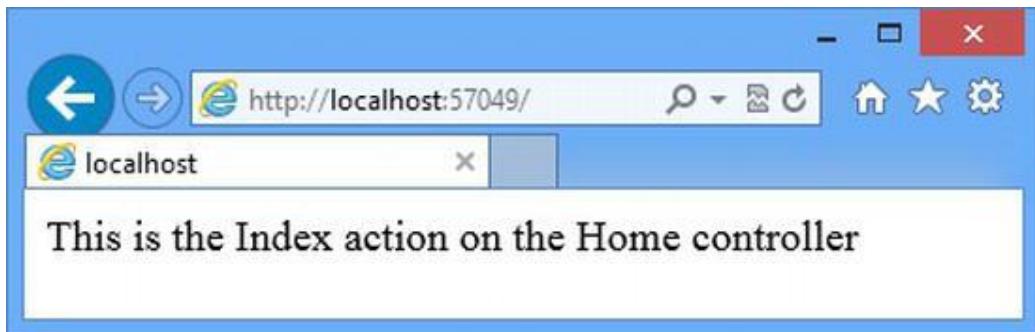
Для этой главы мы создали новый проект MVC под названием `Filters` на шаблоне `Empty`. Мы создали контроллер `Home` с методами действий, которые показаны в листинге 16-5. В этой главе мы сосредоточены только на контроллерах, так что из методов действий будем возвращать строковые значения, а не объекты `ActionResult`. Из-за этого MVC Framework будет отправлять строковые значения непосредственно в браузер, минуя движок представлений Razor. (Мы сделали это для простоты, но предполагая, что в реальных проектах вы будете использовать представления).

#### Листинг 16-5: Контроллер `Home` в проекте `Filters`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
namespace Filters.Controllers
{
    public class HomeController : Controller
    {
        public string Index()
        {
            return "This is the Index action on the Home controller";
        }
    }
}
```

Если вы запустите приложение, то по стандартным маршрутам, определенным Visual Studio, запрошенная нашим браузером ссылка / будет соотнесена с методом действия `Index` контроллера `Home`. Вы увидите результат, показанный на рисунке 16-1.

Рисунок 16-1: Запускаем приложение



## Использование фильтров авторизации

Фильтры авторизации - это фильтры, которые запускаются первыми, перед любым другим фильтром или методом действия. Как следует из названия, эти фильтры осуществляют вашу политику авторизации, гарантируя, что методы действий могут быть вызваны только пользователями, имеющими право доступа. Фильтры авторизации реализуют интерфейс `IAuthorizationFilter`, показанный в листинге 16-6.

Листинг 16-6: Интерфейс `IAuthorizationFilter`

```
namespace System.Web.Mvc
{
    public interface IAuthorizationFilter
    {
        void OnAuthorization(AuthorizationContext filterContext);
    }
}
```

Если хотите, вы можете создать класс, реализующий интерфейс `IAuthorizationFilter`, и написать свою собственную логику безопасности. Однако мы хотим вас предупредить, что это очень плохая идея.

### Предупреждение. Не пишите свой код безопасности!

Мы никогда не пишем собственный код безопасности. История программирования полна обломков приложений, разработчики которых думали, что могут написать хороший код безопасности. На самом деле, это - очень редкий навык. Как правило, остается какая-нибудь забытая или непротестированная мелочь, которая и становится зияющей дырой в безопасности приложения. Если вы нам не верите, просто загуглите фразу *ошибка безопасности* (или *security bug*) и просмотрите топ-результаты.

Везде, где это возможно, мы будем использовать хорошо протестированный и надежный код безопасности. Для таких случаев MVC Framework предоставляет полнофункциональный фильтр авторизации, который можно наследовать, чтобы реализовать пользовательские правила авторизации. Мы стараемся его использовать каждый раз, когда он нам подходит, и рекомендуем вам делать то же самое. По крайней мере, если закрытые данные вашего приложения распространяться по всему интернету, вы сможете переложить часть вины на Microsoft.

Более безопасный подход - создать подкласс класса `AuthorizeAttribute`, который будет содержать самый сложный код и облегчит написание пользовательского кода авторизации. Чтобы это продемонстрировать, мы создадим пользовательский фильтр. Добавим в проект папку

Infrastructure и создадим в нем класс под названием CustomAuthAttribute.cs. Содержимое этого файла показано в листинге 16-7.

### Листинг 16-7: Пользовательский фильтр авторизации

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
namespace Filters.Infrastructure
{
    public class CustomAuthAttribute : AuthorizeAttribute
    {
        private bool localAllowed;
        public CustomAuthAttribute(bool allowedParam)
        {
            localAllowed = allowedParam;
        }
        protected override bool AuthorizeCore(HttpContextBase httpContext)
        {
            if (httpContext.Request.IsLocal)
            {
                return localAllowed;
            }
            else
            {
                return true;
            }
        }
    }
}
```

Это простой фильтр авторизации. Он позволяет блокировать доступ к локальным запросам (локальный запрос осуществляется, когда браузер и сервер приложений запущены на одном устройстве; например, это компьютер, на котором вы разрабатываете приложение).

Мы использовали самый простой подход к созданию фильтра авторизации, который заключается в создании подкласса класса AuthorizeAttribute и переопределении метода AuthorizeCore. Благодаря этому мы сможем пользоваться функциями,строенными в AuthorizeAttribute. Конструктор нашего фильтра принимает значение bool, указывающее, разрешены ли локальные запросы.

Самое интересное в нашем классе фильтра касается реализации метода AuthorizeCore, с помощью которого MVC Framework проверяет, авторизирует ли фильтр доступ для запроса. Аргументом этого метода является объект HttpContextBase, через который мы получаем информацию об обработке запроса. Используя преимущества встроенных функций базового класса AuthorizeAttribute, мы можем сосредоточиться на логике авторизации и вернуть из метода AuthorizeCore true, если хотим авторизовать запрос, и false, если нет.

## Упрощаем атрибуты авторизации

В метод AuthorizeCore передается объект HttpContextBase, который обеспечивает доступ к информации о запросе, но не о контроллере или методе действия, к которому был применен атрибут авторизации. Основная причина, по которой разработчики реализуют интерфейс IAuthorizationFilter напрямую, - это получить доступ к AuthorizationContext, который передается в метод OnAuthorization. Из него можно получить гораздо больше информации, в том числе о маршрутизации и текущем контроллере и методе действия.

Мы не рекомендуем этот подход, и не только потому, что считаем написание собственного кода безопасности опасным. Хотя авторизация и является сквозной функциональностью, но включение логики в атрибуты авторизации, которая тесно связана со структурой контроллеров, нарушает принцип разделения обязанностей, вызывает проблемы с тестированием и поддержкой. Пусть лучше атрибуты авторизации будут простыми и сфокусированными на запросах, а информация о контроллере или методе, к которому применяется авторизация, поставляется из этого контроллера или метода.

## Применяем пользовательский фильтр авторизации

Чтобы использовать пользовательский фильтр авторизации, мы просто применяем атрибут к методам действий или контроллерам, которые мы хотим защитить, как показано в листинге 16-8. Здесь показано применение фильтра к методу действия Index контроллера Home.

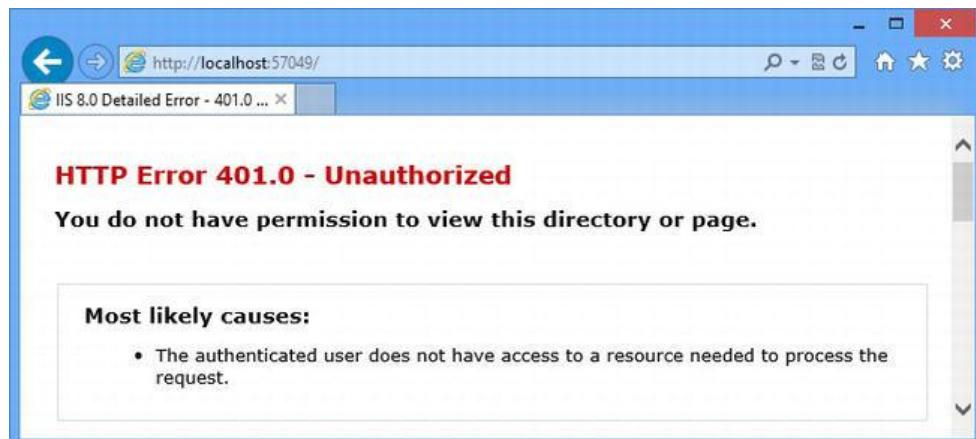
**Листинг 16-8:** Применяем пользовательский фильтр авторизации

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using Filters.Infrastructure;

namespace Filters.Controllers
{
    public class HomeController : Controller
    {
        [CustomAuth(false)]
        public string Index()
        {
            return "This is the Index action on the Home controller";
        }
    }
}
```

Мы установили значение `false` для аргумента конструктора, что означает, что локальным запросам будет отказано в доступе к методу действия `Index`. Чтобы проверить это, запустите приложение - когда браузером будет запрошен корневой URL, конфигурация маршрутизации направит нас на метод действия `Index`. Если отправляющий запрос браузер работает на той же машине, что и Visual Studio, то вы увидите результат, изображенный на рисунке 16-2. Фильтр авторизует запрос, если он поступит от браузера, работающего на другом компьютере (или если мы присвоим аргументу конструктора фильтра значение `true` и перезапустим приложение).

**Рисунок 16-2:** Пользовательский фильтр авторизации отказывает в доступе локальным запросам



## Используем встроенный фильтр авторизации

Хотя мы использовали класс `AuthorizeAttribute` как основу для пользовательского фильтра, у него есть своя собственная реализация метода `AuthorizeCore`, который используется для выполнения общих задач авторизации.

Используя `AuthorizeAttribute` напрямую, мы можем определить правила авторизации с помощью двух доступных свойств этого класса, как показано в таблице 16-2.

**Таблица 16-2:** Свойства `AuthorizeAttribute`

Название	Тип	Описание
Users	string	Разделенный запятыми список имен пользователей, которым разрешен доступ к методу действия.
Roles	string	Разделенный запятыми список названий ролей. Чтобы получить доступ к методу действия, пользователь должен обладать по крайней мере одной из этих ролей.

В листинге 16-9 показано, как можно использовать эти встроенные фильтры и их свойства для защиты метода действия.

**Листинг 16-9:** Используем встроенный фильтр авторизации

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using Filters.Infrastructure;
namespace Filters.Controllers
{
    public class HomeController : Controller
    {
        [Authorize(Users = "adam, steve, jacqui", Roles = "admin")]
        public string Index()
        {
            return "This is the Index action on the Home controller";
        }
    }
}
```

В листинге мы указали как пользователей, так и роли. Это означает, что авторизация будет пройдена, только если оба условия будут выполнены: имя пользователя - `adam`, `steve` или `jacqui` и пользователь является администратором. По умолчанию, каждый запрос будет аутентифицирован. Если мы не указываем имена пользователей или роли для аутентификации, то любой авторизованный пользователь сможет использовать метод действия.

### Подсказка

*`AuthorizeAttribute` отвечает за авторизацию, но не за аутентификацию. Вы можете использовать любую из встроенных систем аутентификации ASP.NET или разработать свою собственную (хотя, как отмечалось ранее, это не очень хорошая идея). Пока система аутентификации использует стандартные API ASP.NET, `AuthorizeAttribute` сможет ограничивать доступ к контроллерам и действиям.*

*Для большинства приложений достаточно правил авторизации, которые обеспечивает `AuthorizeAttribute`. Если вы хотите реализовать что-то особенное, то можете наследовать этот класс, как мы делали ранее в этой главе.*

# Использование фильтров для исключений

Если при вызове метода действия было выброшено необработанное исключение, будет запущен фильтр исключений. Исключения могут поступать из:

- другого фильтра (фильтра авторизации, действия или результата);
- самого метода действия;
- при выполнении результата действия (подробная информация о результатах действия дана в главе 15).

## Создаем фильтр исключения

Фильтры исключений должны реализовывать интерфейс `IExceptionFilter`, который показан в листинге 16-10.

**Листинг 16-10:** Интерфейс `IExceptionFilter`

```
namespace System.Web.Mvc
{
    public interface IExceptionFilter
    {
        void OnException(ExceptionContext filterContext);
    }
}
```

Когда появится необработанное исключение, будет вызван метод `OnException`. Параметром для этого метода является объект `ExceptionContext`, который наследует от `ControllerContext` и имеет ряд полезных свойств, с помощью которых можно получить информацию о запросе. Они приведены в таблице 16-3.

**Таблица 16-3:** Свойства `ControllerContext`

Название	Тип	Описание
Controller	ControllerBase	Возвращает объект контроллера для данного запроса
HttpContext	HttpContextBase	Обеспечивает доступ к информации о запросе и доступ к ответу
IsChildAction	bool	Возвращает <code>true</code> , если это дочернее действие (будет кратко обсуждаться позже в этой главе и подробно в главе 18)
RequestContext	RequestContext	Предоставляет доступ к объекту <code>HttpContext</code> и данным маршрутизации, хотя и то, и то доступно через другие свойства
RouteData	RouteData	Возвращает данные маршрутизации для данного запроса

В дополнение к свойствам, наследованным от класса `ControllerContext`, класс `ExceptionContext` определяет некоторые дополнительные свойства, которые также полезны при работе с исключениями. Они показаны в таблице 16-4.

### Подсказка

*Мы разделили эти таблицы, потому что есть и другие классы контекста фильтров, которые наследуют от ControllerContext и которые мы представим по ходу этой главы.*

**Таблица 16-4:** Дополнительные свойства ExceptionContext

Название	Тип	Описание
ActionDescriptor	ActionDescriptor	Предоставляет подробную информацию о методе действия
Result	ActionResult	Результат для метода действия; фильтр может отменить запрос, установив для этого свойства иное значение, кроме null
Exception	Exception	Необработанное исключение
ExceptionHandled	bool	Возвращает true, если другой фильтр отметил это исключение как обработанное

Сгенерированное исключение доступно через свойство Exception. Фильтр исключения может сообщить, что он обработал исключение, установив для свойства ExceptionHandled значение true. Вызываются все фильтры исключений, примененные к действию, даже если ExceptionHandled уже содержит true, так что лучше всегда проверять, обработано ли исключение другим фильтром, чтобы не решать повторно уже решенную проблему.

#### Примечание

*Если ни один из фильтров исключений не установил свойству ExceptionHandled значение true, MVC Framework будет использовать стандартную процедуру обработки исключений ASP.NET, которая приведет к самому нежелательному результату - "желтому экрану смерти".*

Свойство Result используется фильтром исключений, чтобы сообщить MVC Framework дальнейшую последовательность действий. В основном это регистрация исключения и отображение соответствующего сообщения для пользователя. Чтобы продемонстрировать выполнение этих задач, мы создали новый класс под названием RangeExceptionAttribute.cs, который добавили в папку Infrastructure нашего проекта. Содержимое этого файла показано в листинге 16-11.

**Листинг 16-11:** Реализуем фильтр исключений

```
using System;
using System.Web.Mvc;
namespace Filters.Infrastructure
{
    public class RangeExceptionAttribute : FilterAttribute, IExceptionFilter
    {
        public void OnException(ExceptionContext filterContext)
        {
            if (!filterContext.ExceptionHandled &&
                filterContext.Exception is ArgumentOutOfRangeException)
            {
                filterContext.Result
                    = new RedirectResult("~/Content/RangeErrorPage.html");
                filterContext.ExceptionHandled = true;
            }
        }
    }
}
```

Этот фильтр исключений обрабатывает экземпляры `ArgumentOutOfRangeException`, перенаправляя браузер пользователя к файлу `RangeErrorPage.html` из папки `Content`.

Обратите внимание, что мы наследовали класс `RangeExceptionAttribute` от класса `FilterAttribute` наряду с реализацией интерфейса `IExceptionFilter`. Чтобы класс атрибута .NET работал как фильтр MVC, класс должен реализовать интерфейс `IMvcFilter`. Это можно сделать напрямую, но самый простой способ создать фильтр – это наследовать от класса `FilterAttribute`, который реализует необходимый интерфейс и предоставляет некоторые полезные базовые функции, такие как изменение стандартного порядка выполнения фильтров (к чему мы вернемся позже в этой главе).

## Применяем фильтр исключений

Прежде чем мы сможем протестировать наш фильтр исключений, нужно создать для этого некоторую базу. Во-первых, создадим в проекте папку `Content` и в ней - файл `RangeErrorPage.html`. Он будет использоваться для отображения простого сообщения, которые вы можете видеть в листинге 16-12.

**Листинг 16-12:** Содержимое файла `RangeErrorPage.html`

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Range Error</title>
</head>
<body>
    <h2>Sorry</h2>
    <span>One of the arguments was out of the expected range.</span>
</body>
</html>
```

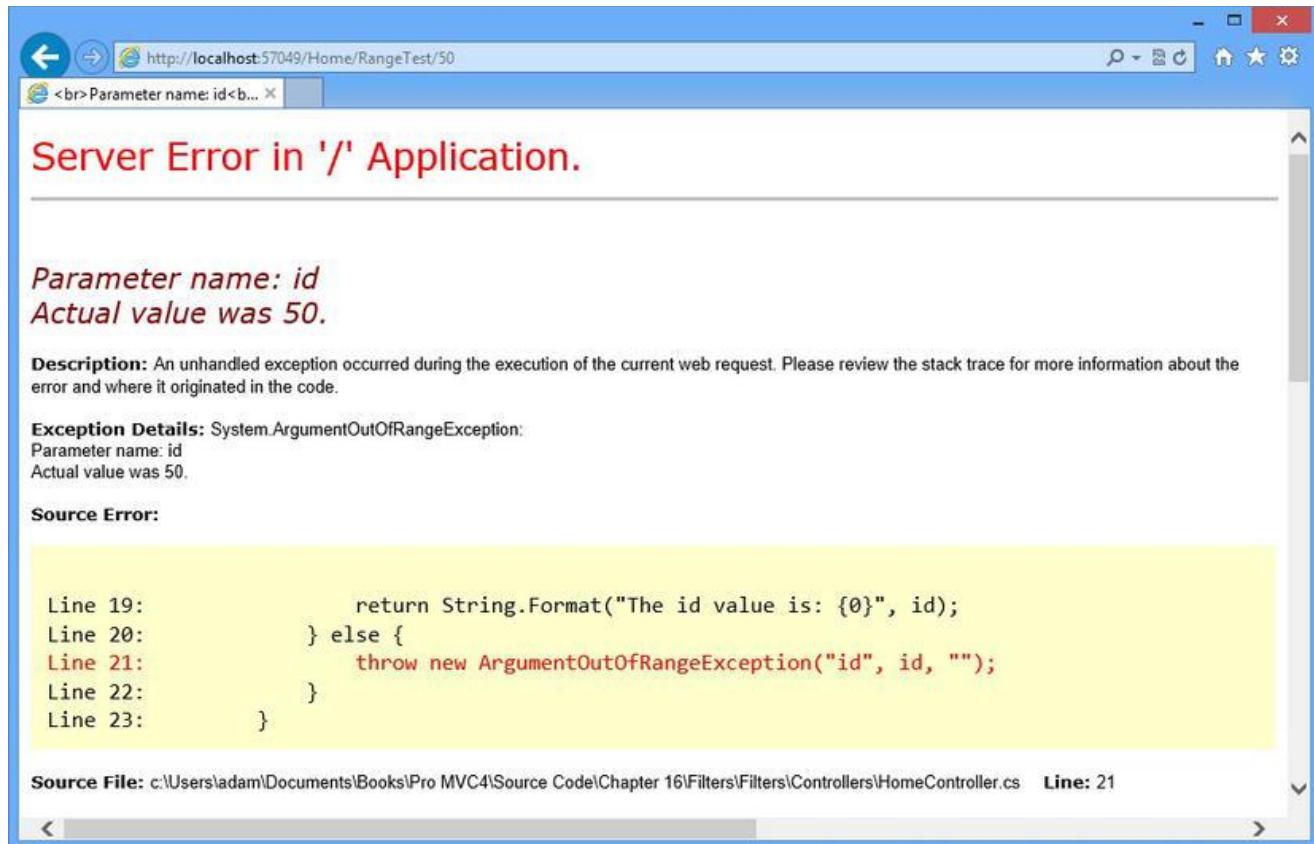
Далее нам нужно добавить метод действия в контроллер `Home`, который будет выбрасывать интересующее нас исключение. Он показан в листинге 16-13.

**Листинг 16-13:** Добавляем новое действие в контроллер `Home`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using Filters.Infrastructure;
namespace Filters.Controllers
{
    public class HomeController : Controller
    {
        [Authorize(Users = "adam, steve, jacqui", Roles = "admin")]
        public string Index()
        {
            return "This is the Index action on the Home controller";
        }
        public string RangeTest(int id)
        {
            if (id > 100)
            {
                return String.Format("The id value is: {0}", id);
            }
            else
            {
                throw new ArgumentOutOfRangeException("id", id, "");
            }
        }
    }
}
```

Если вы запустите приложение и перейдете по ссылке /Home/RangeTest/50, то увидите стандартную обработку исключений. В маршрутизации по умолчанию, которую для проекта создает Visual Studio, есть переменная сегмента под названием id, которой в этом URL мы установили значение 50, что и приведет к результату, показанному на рисунке 16-3. (Подробно маршрутизация и сегменты URL описаны в главах 13 и 14.)

**Рисунок 16-3:** Ответ стандартной обработки исключений



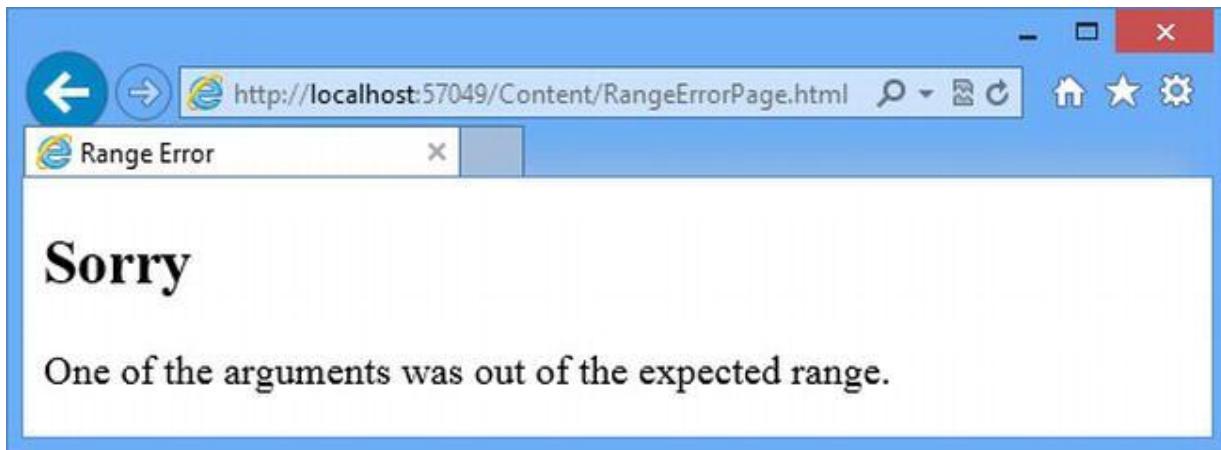
Мы можем применить фильтр исключений либо к контроллерам, либо к отдельным действиям, как показано в листинге 16-14.

**Листинг 16-14:** Применяем фильтр

```
[RangeException]
public string RangeTest(int id)
{
    if (id > 100)
    {
        return String.Format("The id value is: {0}", id);
    }
    else
    {
        throw new ArgumentOutOfRangeException("id");
    }
}
```

Если вы перезапустите приложение и снова перейдете по ссылке Home/RangeTest/50, то увидите результат, показанный на рисунке 16-4.

**Рисунок 16-4:** Эффект применения фильтра исключений



### Используем представление для вывода ответа при выбросе исключения

Проще и безопаснее всего для отображения ответа при выбросе исключения использовать страницу или статический контент – едва ли при таком подходе что-то пойдет не так и возникнут дополнительные проблемы. Однако такое решение будет не достаточно хорошим для пользователя, который получит общее предупреждение и будет выброшен из приложения.

Альтернативный подход – использовать представление для отображения конкретного сообщения о проблеме и предоставления пользователю некоторой контекстной информации и возможностей, с помощью которых можно данную проблему решить. Чтобы это продемонстрировать, мы внесли некоторые изменения в класс RangeExceptionAttribute, как показано в листинге 16-15.

### Листинг 16-15: Возвращаем представление из фильтра исключений

```
using System;
using System.Web.Mvc;
namespace Filters.Infrastructure
{
    public class RangeExceptionAttribute : FilterAttribute, IExceptionFilter
    {
        public void OnException(ExceptionContext filterContext)
        {
            if (!filterContext.ExceptionHandled &&
                filterContext.Exception is ArgumentOutOfRangeException)
            {
                int val = (int)((ArgumentOutOfRangeException)
filterContext.Exception).ActualValue;
                filterContext.Result = new ViewResult
                {
                    ViewName = "RangeError",
                    ViewData = new ViewDataDictionary<int>(val)
                };
                filterContext.ExceptionHandled = true;
            }
        }
    }
}
```

Мы создали объект ViewResult и установили значения его свойств ViewName и ViewData, определяющие название представления и объект модели, который будет в него передан. Это немного запутанный код, потому что мы работаем с объектом ViewResult напрямую и не используем определенный в классе Controller метод View, который всегда применяем в методах действий. Мы не станем останавливаться на этом коде, потому что представления будут подробно рассмотрены в

главе 18, и встроенный фильтр исключений, который мы опишем в следующем разделе, позволит достичь того же эффекта более понятным способом. Сейчас мы просто хотим вам показать, как все работает «под капотом».

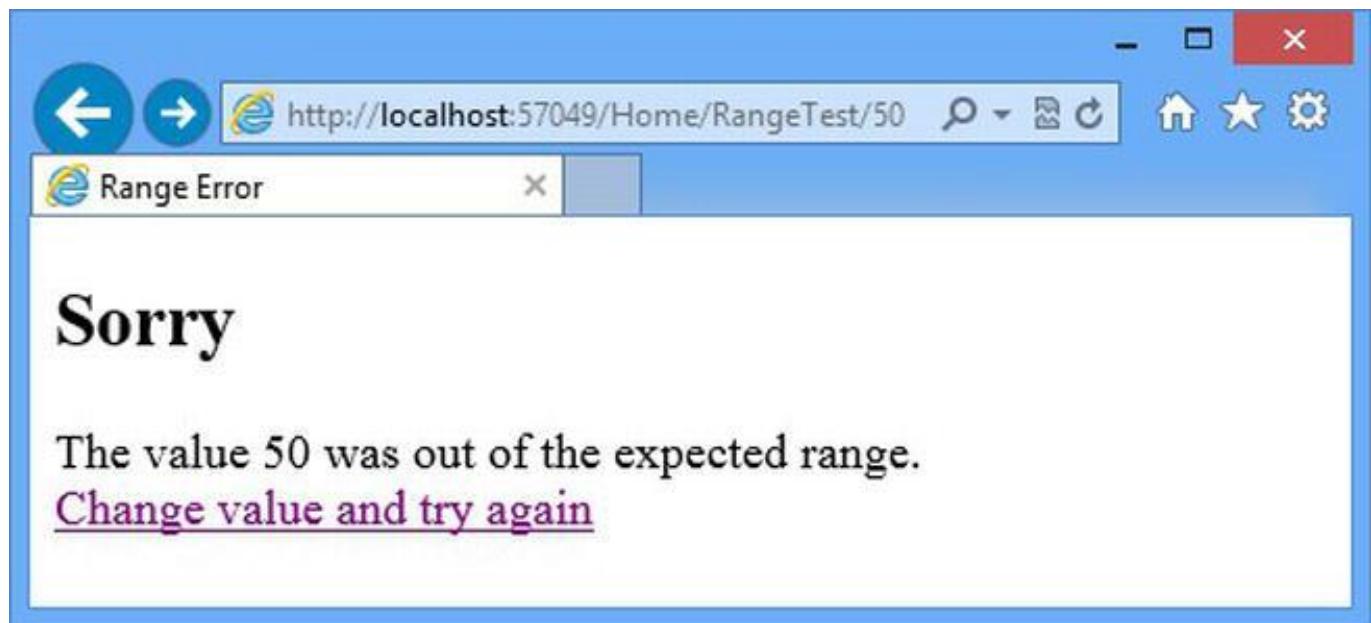
В объекте `ViewResult` мы указываем представление под названием `RangeError` и передаем значение `int` вызвавшего исключение аргумента в качестве объекта модели представления. Затем мы добавим папку `Views/Shared` в проект Visual Studio и создадим в нем файл `RangeError.cshtml`, содержимое которого показано в листинге 16-16.

**Листинг 16-16:** Файл представления `RangeError.cshtml`

```
@model int
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Range Error</title>
</head>
<body>
    <h2>Sorry</h2>
    <span>The value @Model was out of the expected range.</span>
    <div>
        @Html.ActionLink("Change value and try again", "Index")
    </div>
</body>
</html>
```

В файле представления мы используем стандартные теги HTML и Razor, чтобы предоставить пользователю (немного) более полезное сообщение. Наш пример приложения довольно ограничен, поэтому у нас нет каких-либо полезных страниц, на которые можно направить пользователя для решения этой проблемы. Мы создали ссылки на другой метод действия с помощью вспомогательного метода `ActionLink`, просто чтобы показать, что в представлении вы можете использовать какие угодно функции. Чтобы увидеть результат, перезапустите приложение и перейдите по ссылке `/Home/RangeTest/50`, как показано на рисунке 16-5.

**Рисунок 16-5:** Используем представление для отображения сообщения от фильтра исключения



## Как избегать лишних исключений

Преимущества использования представления для отображения ошибки заключаются в том, что вы можете использовать макеты, чтобы сообщение об ошибке выглядело согласовано со всем приложением, и генерировать динамический контент, который поможет пользователю понять, что произошло не так и что можно исправить.

Недостатком этого подхода является то, что вам придется тщательно протестировать представление и убедиться, что вы не просто генерируете еще одно исключение. Мы часто с этим сталкиваемся, когда разработчики уделяют основное внимание тестированию основных функций приложения и не учитывают всех потенциальных ситуаций, которые могут привести к ошибке. Чтобы продемонстрировать это, мы добавили блок кода Razor в представление RangeError.cshtml, который вызовет исключение, как показано в листинге 16-17.

**Листинг 16-17:** Добавляем в представление код, который вызовет исключение

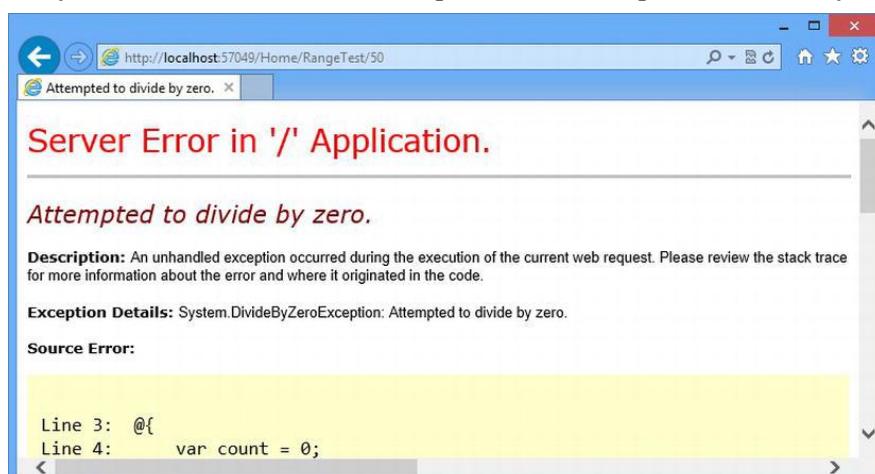
```
@model int

{@
    var count = 0;
    var number = Model / count;
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Range Error</title>
</head>
<body>
    <h2>Sorry</h2>
    <span>The value @Model was out of the expected range.</span>
    <div>
        @Html.ActionLink("Change value and try again", "Index")
    </div>
</body>
</html>
```

При визуализации представления наш код будет генерировать DivideByZeroException. Если вы запустите приложение и снова перейдите по ссылке /Home/RangeTest/50, то увидите исключение, которое возникает при попытке визуализировать представление, а не то, которое выбрасывает контроллер, как показано на рисунке 16-6.

**Рисунок 16-6:** Исключение, которое возникает при попытке визуализировать представление



Это нереалистичный сценарий, но он показывает, что произойдет, если у нас будут проблемы с представлением: пользователь увидит странную ошибку, которая даже не относится к его проблеме с приложением. Если вы используете представление для фильтра исключений, тщательно протестируйте это представление.

## Используем встроенный фильтр исключений

Мы разобрали процесс создания фильтров исключений, потому что важно понимать, что происходит «под капотом» в MVC Framework. Но в реальных проектах вам не часто понадобится создавать пользовательские фильтры, потому что Microsoft включила в MVC Framework атрибут `HandleErrorAttribute`, который является встроенной реализацией интерфейса `IExceptionFilter`. Вы можете указать исключения, имена представлений и разметку, используя свойства, описанные в таблице 16-5.

**Таблица 16-5:** Свойства `HandleErrorAttribute`

Название	Тип	Описание
<code>ExceptionType</code>	<code>Type</code>	Тип исключения, который обрабатывается данным фильтром. Это свойство также будет обрабатывать типы исключений, которые наследуют от указанного, но будет игнорировать все другие. По умолчанию для <code>ExceptionType</code> указано значение <code>System.Exception</code> , что означает, что оно будет обрабатывать все стандартные исключения.
<code>View</code>	<code>string</code>	Название шаблона представления, которое визуализируется данным фильтром. Если вы не указываете значение, по умолчанию устанавливается значение <code>Error</code> , так что будет визуализировано <code>Views/&lt;currentControllerName&gt;/Error.cshtml</code> или <code>/Views/Shared/Error.cshtml</code> .
<code>Master</code>	<code>string</code>	Имя макета, который используется при визуализации представления данного фильтра. Если вы не указываете значение, представление использует свой макет страницы по умолчанию.

Когда появляется необработанное исключение указанного типа в `ExceptionType`, `HandleErrorAttribute` визуализирует представление, указанное в свойстве `View` (используя макет по умолчанию или определенный в свойстве `Master`).

## Готовимся использовать встроенный фильтр исключений

Фильтр `HandleErrorAttribute` работает только тогда, когда в файле `Web.config` включена обработка пользовательских исключений. Поэтому мы добавляем атрибут `customErrors` в узел `<system.web>`, как показано в листинге 16-18.

**Листинг 16-18:** Включаем обработку пользовательских исключений в файле `Web.config`

```
<system.web>
  <httpRuntime targetFramework="4.5" />
  <compilation debug="true" targetFramework="4.5" />
  <pages>
    <namespaces>
      <add namespace="System.Web.Helpers" />
      <add namespace="System.Web.Mvc" />
      <add namespace="System.Web.Mvc.Ajax" />
      <add namespace="System.Web.Mvc.Html" />
      <add namespace="System.Web.Routing" />
      <add namespace="System.Web.WebPages" />
    </namespaces>
  </pages>
</system.web>
```

```

</pages>
<customErrors mode="On" defaultRedirect="/Content/RangeErrorPage.html"/>
</system.web>

```

По умолчанию для атрибута `mode` установлено значение `RemoteOnly`, что означает, что во время разработки `HandleErrorAttribute` не будет перехватывать исключения, но когда вы развернете приложение на сервере и начнете делать запросы с другого компьютера, то `HandleErrorAttribute` вступит в силу. Чтобы понять, что увидят конечные пользователи, установите для `customErrors` значение `On`. В атрибуте `defaultRedirect` указывается страница по умолчанию, которая будет отображаться, если все остальные не работают.

## Применяем встроенный фильтр исключений

В листинге 16-19 показано, как мы применили фильтр `HandleError` к контроллеру `Home`.

**Листинг 16-19:** Используем фильтр `HandleErrorAttribute`

```

[HandleError(ExceptionType = typeof(ArgumentOutOfRangeException), View = "RangeError")]
public string RangeTest(int id)
{
    if (id > 100)
    {
        return String.Format("The id value is: {0}", id);
    }
    else
    {
        throw new ArgumentOutOfRangeException("id", id, "");
    }
}

```

В этом примере мы создали ту же ситуацию, которая у нас была с пользовательским фильтром, то есть при возникновении `ArgumentOutOfRangeException` пользователь увидит представление `RangeError`.

Визуализируя представление, фильтр `HandleErrorAttribute` передает объект модели представления `HandleErrorInfo`, который содержит исключение и дополнительную информацию, которую мы будем использовать в представлении. В таблице 16-6 приведены свойства, определенные в классе `HandleErrorInfo`.

**Таблица 16-6:** Свойства `HandleErrorInfo`

Название	Тип	Описание
ActionName	string	Возвращает имя метода действия, который сгенерировал исключение
ControllerName	string	Возвращает имя контроллера, который сгенерировал исключение
Exception	Exception	Возвращает исключение

Обратите внимание, как мы обновили представление `RangeError.cshtml`, чтобы использовать этот объект модели в листинге 16-20.

**Листинг 16-20:** Используем объект модели `HandleErrorInfo` в представлении `RangeError`

```

@model HandleErrorInfo
 @{
    ViewBag.Title = "Sorry, there was a problem!";
}
<!DOCTYPE html>
<html>

```

```

<head>
    <meta name="viewport" content="width=device-width" />
    <title>Range Error</title>
</head>
<body>
    <h2>Sorry</h2>
    <span>The value @(((ArgumentOutOfRangeException)Model.Exception).ActualValue)
        was out of the expected range.</span>
    <div>
        @Html.ActionLink("Change value and try again", "Index")
    </div>
    <div style="display: none">
        @Model.Exception.StackTrace
    </div>
</body>
</html>

```

Мы должны были привести значение свойства `Model.Exception` к типу `ArgumentOutOfRangeException` только затем, чтобы потом иметь возможность прочитать свойство `ActualValue`, так как класс `HandleErrorInfo` является объектом модели общего назначения и используется для передачи любого исключения в представление.

#### *Внимание*

*При использовании фильтра HandleError иногда возникает странное поведение, при котором представление не отображается пользователю, пока в него не включено значение свойства Model.Exception.StackTrace. Так как мы не хотим его отображать, мы заключили его вывод в элемент div, в котором свойству CSS display задали значение none, благодаря чему он стал невидимым для пользователя.*

## Использование фильтров для методов действий

Фильтры действий можно использовать для различных целей. Встроенный класс для создания таких фильтров, `IActionFilter`, показан в листинге 16-21.

#### Листинг 16-21: Интерфейс `IActionFilter`

```

namespace System.Web.Mvc
{
    public interface IActionFilter
    {
        void OnActionExecuting(ActionExecutingContext filterContext);
        void OnActionExecuted(ActionExecutedContext filterContext);
    }
}

```

Этот интерфейс определяет два метода. MVC Framework вызывает метод `OnActionExecuting` перед тем, как вызвать метод действия, и `OnActionExecuted` после вызова метода действия.

#### Реализуем метод `OnActionExecuting`

Метод `OnActionExecuting` вызывается до метода действия. Вы можете использовать его для того, чтобы изучить запрос и принять решение о его отмене, изменении или задержке его выполнения. Параметром этого метода является объект `ActionExecutingContext`, который создает подкласс класса `ControllerContext` и определяет два дополнительных свойства, описанных в таблице 16-7.

**Таблица 16-7:** Свойства ActionExecutingContext

Название	Тип	Описание
ActionDescriptor	ActionDescriptor	Предоставляет информацию о методе действия
Result	ActionResult	Результат для метода действия; фильтр может отменить запрос, установив данному свойству иное значение, кроме null

Вы можете использовать фильтр, чтобы отменить запрос, определив в качестве значения для свойства `Result` результат действия. Чтобы это продемонстрировать, мы создали свой собственный класс фильтра действий под названием `CustomActionAttribute` в папке `Infrastructure`, как показано в листинге 16-22.

#### Листинг 16-22: Отмена запроса в методе OnActionExecuting

```
using System.Web.Mvc;
namespace Filters.Infrastructure
{
    public class CustomActionAttribute : FilterAttribute, IActionFilter
    {
        public void OnActionExecuting(ActionExecutingContext filterContext)
        {
            if (filterContext.HttpContext.Request.IsLocal)
            {
                filterContext.Result = new HttpNotFoundResult();
            }
        }
        public void OnActionExecuted(ActionExecutedContext filterContext)
        {
            // not yet implemented
        }
    }
}
```

В этом примере мы используем метод `OnActionExecuting`, чтобы проверить, отправлен ли запрос с локальной машины. Если это так, то возвращаем пользователю ответ 404 – Not Found.

#### Примечание

*Как вы видите из листинга 16-22, необязательно реализовывать оба метода, определенные в интерфейсе `IActionFilter`, чтобы создать рабочий фильтр. Будьте осторожны, чтобы не вызвать `NotImplementedException`, которое Visual Studio добавляет к классу при реализации данного интерфейса. В фильтре действий MVC Framework вызывает оба метода, и если возникает исключение, то будут инициированы фильтры исключений. Если вам не нужно добавлять к методу какую-либо логику, просто оставьте его пустым.*

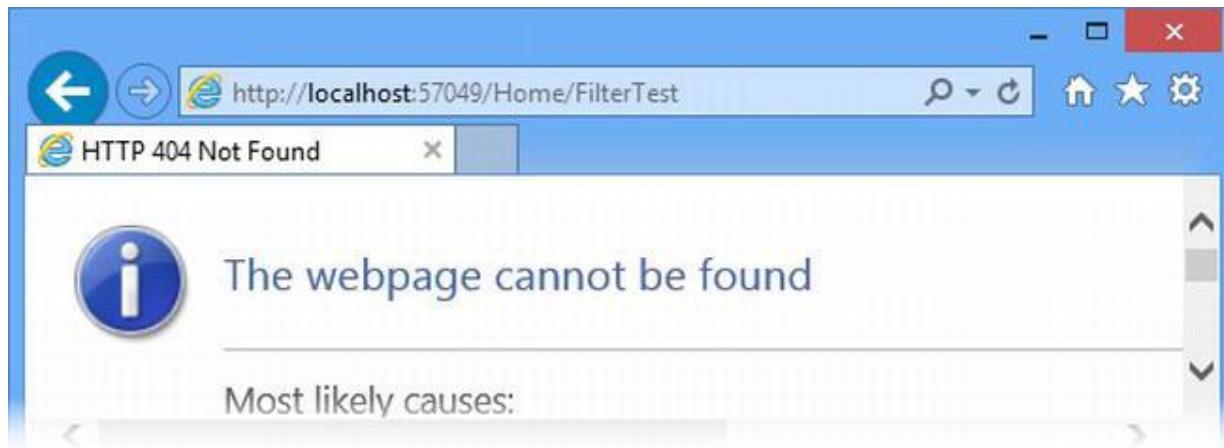
Фильтр действий применяется так же, как и любой другой атрибут. Чтобы продемонстрировать фильтр, который мы создали в листинге 16-22, мы добавили новый метод действия в контроллер `Home`, как показано в листинге 16-23.

#### Листинг 16-23: Добавляем новое действие в контроллер Home

```
[CustomAction]
public string FilterTest()
{
    return "This is the FilterTest action";
}
```

Вы можете протестировать фильтр, запустив приложение и перейдя по ссылке /Home/FilterTest. Если вы отправили запрос с локальной машины, то вы увидите результат, показанный на рисунке 16-7, хотя, как мы знаем, и контроллер, и действие существуют.

**Рисунок 16-7:** Эффект от использования фильтра действий



## Реализуем метод OnActionExecuted

Фильтр также можно использовать для выполнения каких-либо задач, которые измеряют время выполнения метода действия. В качестве простого примера мы создали новый класс `ProfileActionAttribute` в папке `Infrastructure`, который измеряет количество времени, которое занимает выполнение метода действия. Код этого фильтра показан в листинге 16-24.

### Листинг 16-24: Более сложный фильтр действий

```
using System.Diagnostics;
using System.Web.Mvc;
namespace Filters.Infrastructure
{
    public class ProfileActionAttribute : FilterAttribute, IActionFilter
    {
        private Stopwatch timer;
        public void OnActionExecuting(ActionExecutingContext filterContext)
        {
            timer = Stopwatch.StartNew();
        }
        public void OnActionExecuted(ActionExecutedContext filterContext)
        {
            timer.Stop();
            if (filterContext.Exception == null)
            {
                filterContext.HttpContext.Response.Write(
                    string.Format("<div>Action method elapsed time: {0}</div>",
                    timer.Elapsed.TotalSeconds));
            }
        }
    }
}
```

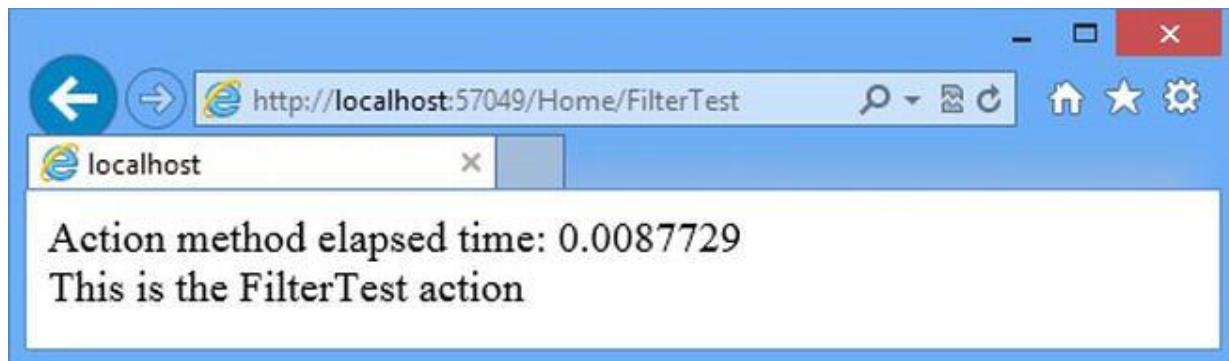
В этом примере мы запускаем таймер с помощью метода `OnActionExecuting` (используется класс таймера `Stopwatch` из пространства имен `System.Diagnostics`). Метод `OnActionExecuted` вызывается по завершении метода действия. В листинге 16-25 показано, как мы применили атрибут к контроллеру `Home` (ранее созданный фильтр был удален, чтобы локальные запросы не перенаправлялись).

### Листинг 16-25: Применяем фильтр действий к контроллеру Home

```
[ProfileAction]
public string FilterTest()
{
    return "This is the ActionFilterTest action";
}
```

Если вы запустите приложение и перейдете по ссылке /Home/FilterTest, то увидите те же результаты, что и на рисунке 16-8.

**Рисунок 16-8:** Используем фильтр действий для измерения производительности



*Обратите внимание, что информация о производительности отображается в браузере перед результатом метода действия. Так происходит потому, что фильтр действий выполняется после завершения метода действия, но до обработки результата.*

В качестве параметра в метод OnActionExecuted передается объект ActionExecutedContext. Этот класс определяет дополнительные свойства, которые показаны в таблице 16-8. Свойство Exception возвращает любое исключение, выброшенное методом действия, а свойство ExceptionHandled указывает, было ли оно обработано другим фильтром.

**Таблица 16-8:** Свойства ActionExecutedContext

Название	Тип	Описание
ActionDescriptor	ActionDescriptor	Предоставляет информацию о методе действия
Canceled	bool	Возвращает true, если действие было отменено другим фильтром
Exception	Exception	Возвращает исключение, выброшенное другим фильтром или методом действия
ExceptionHandled	bool	Возвращает true, если исключение было обработано
Result	ActionResult	Результат для метода действия; фильтр может отменить запрос, установив для этого свойства иное значение, кроме null

Свойство Canceled возвращает true, если запрос был отменен другим фильтром (т.е. он установил значение для свойства Result) с того момента, когда был запущен метод фильтра OnActionExecuting. Метод OnActionExecuted все равно будет вызван, но только для того, чтобы можно было освободить или очистить ранее используемые ресурсы.

# Использование фильтров для результата

Фильтры результатов – это фильтры общего назначения, которые обрабатывают результаты, произведенные методами действий. Фильтры результатов реализуют интерфейс `IResultFilter`, который показан в листинге 16-26.

## Листинг 16-26: Интерфейс `IResultFilter`

```
namespace System.Web.Mvc
{
    public interface IResultFilter
    {
        void OnResultExecuting(ResultExecutingContext filterContext);
        void OnResultExecuted(ResultExecutedContext filterContext);
    }
}
```

Как мы узнали в главе 15, методы действий возвращают результаты действий, что позволяет нам отделять цель метода действия от его исполнения. Применение фильтра результатов к методу действия означает, что метод `OnResultExecuting` будет вызван после того, как метод действия вернет результат, но до того, как результат будет выполнен. Метод `OnResultExecuted` будет вызван после выполнения результата действия.

Параметрами для этих методов являются соответственно объекты `ResultExecutingContext` и `ResultExecutedContext`, и они очень похожи на аналогичные фильтры методов действий. Они определяют те же свойства, которые обладают теми же эффектами (см. таблицу 16-8).

Чтобы продемонстрировать простой фильтр результатов, мы создали новый файл класса под названием `ProfileResultAttribute.cs` в папке `Infrastructure` и определили в нем класс, показанный в листинге 16-27.

## Листинг 16-27: Простой фильтр результатов

```
using System.Diagnostics;
using System.Web.Mvc;
namespace Filters.Infrastructure
{
    public class ProfileResultAttribute : FilterAttribute, IResultFilter
    {
        private Stopwatch timer;
        public void OnResultExecuting(ResultExecutingContext filterContext)
        {
            timer = Stopwatch.StartNew();
        }
        public void OnResultExecuted(ResultExecutedContext filterContext)
        {
            timer.Stop();
            filterContext.HttpContext.Response.Write(
                string.Format("<div>Result elapsed time: {0}</div>",
                    timer.Elapsed.TotalSeconds));
        }
    }
}
```

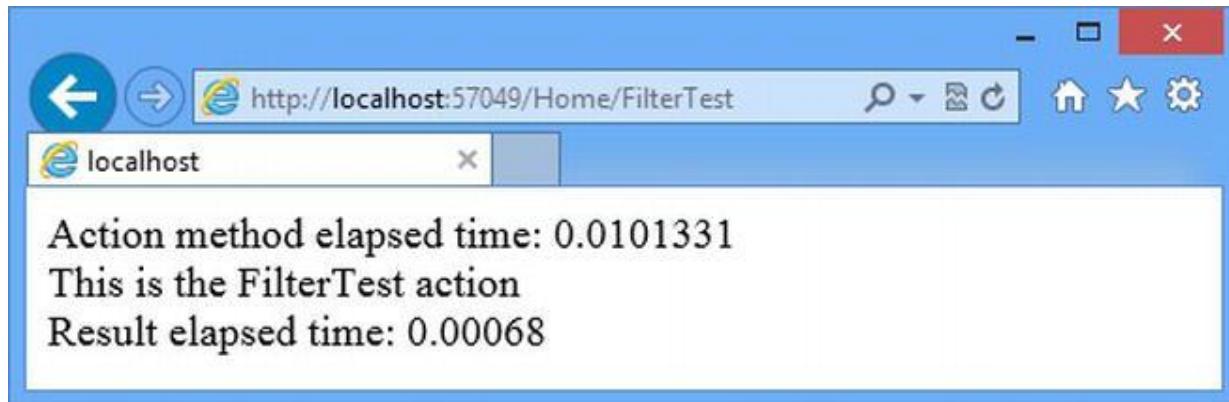
Этот фильтр результатов является дополнением к фильтру действий, который мы создали в предыдущем разделе, и измеряет количество времени, необходимого для выполнения результата. В листинге 16-28 вы можете увидеть, как мы применили этот новый фильтр к контроллеру `Home`.

### Листинг 16-28: Применяем фильтр результатов к контроллеру Home

```
[ProfileAction]
[ProfileResult]
public string FilterTest()
{
    return "This is the ActionFilterTest action";
}
```

На рисунке 16-9 показан результат запуска программы и перехода по ссылке /Home/FilterTest. Как видите, оба фильтра добавили данные в ответ браузеру - вывод из фильтра результата показан после результата от метода действия. Естественно, поскольку MVC Framework не может выполнить метод OnResultExecuted до обработки результата, строковое значение будет вставлено в результат.

**Рисунок 16-9:** Эффект применения фильтра результатов



### Встроенный класс фильтра действий и результатов

MVC Framework включает в себя встроенный класс, который можно использовать для создания как фильтров действий, так и фильтров результатов. Этот класс, `ActionFilterAttribute`, показан в листинге 16-29.

### Листинг 16-29: Класс ActionFilterAttribute

```
public abstract class ActionFilterAttribute : FilterAttribute, IActionFilter,
IResultFilter
{
    public virtual void OnActionExecuting(ActionExecutingContext filterContext)
    {
    }
    public virtual void OnActionExecuted(ActionExecutedContext filterContext)
    {
    }
    public virtual void OnResultExecuting(ResultExecutingContext filterContext)
    {
    }
    public virtual void OnResultExecuted(ResultExecutedContext filterContext)
    {
    }
}
```

Единственное преимущество этого класса заключается в том, что вам не придется переопределять и реализовать методы, которые вы не собираетесь использовать. Никаких других преимуществ реализации интерфейсов фильтров напрямую нет.

Чтобы продемонстрировать работу класса `ActionFilterAttribute`, мы добавили новый файл под названием `ProfileAllAttribute.cs` в папку `Infrastructure` и определили в нем класс, показанный в листинге 16-30.

#### Листинг 16-30: Используем класс `ActionFilterAttribute`

```
using System.Diagnostics;
using System.Web.Mvc;
namespace Filters.Infrastructure
{
    public class ProfileAllAttribute : ActionFilterAttribute
    {
        private Stopwatch timer;
        public override void OnActionExecuting(ActionExecutingContext filterContext)
        {
            timer = Stopwatch.StartNew();
        }
        public override void OnResultExecuted(ResultExecutedContext filterContext)
        {
            timer.Stop();
            filterContext.HttpContext.Response.Write(
                string.Format("<div>Total elapsed time: {0}</div>",
                timer.Elapsed.TotalSeconds));
        }
    }
}
```

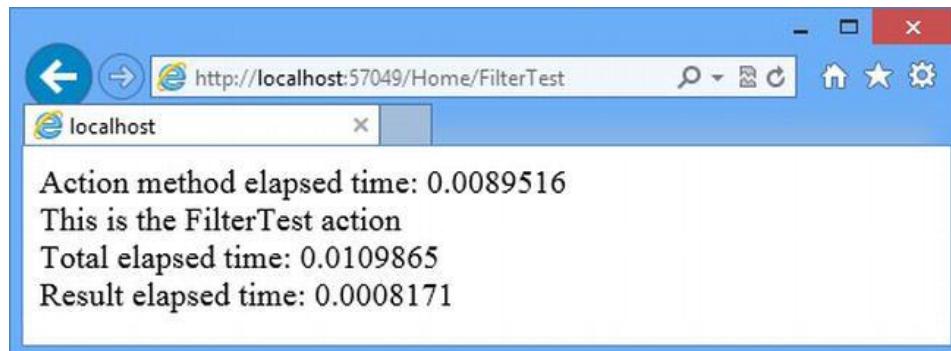
Класс `ActionFilterAttribute` реализует интерфейсы `IActionFilter` и `IResultFilter`, что означает, что MVC Framework будет считать производные классы фильтрами обоих типов, даже если не все методы переопределены. В нашем примере мы реализовали только метод `OnActionExecuting` из интерфейса `IActionFilter` и `OnResultExecuted` из интерфейса `IResultFilter`, что позволит нам продолжить анализ производительности и измерить количество времени, необходимое для выполнения метода действия и обработки результата, в едином блоке. В листинге 16-31 показано применение фильтра к контроллеру `Home`.

#### Листинг 16-31: Применяем фильтр к контроллеру `Home`

```
[ProfileAction]
[ProfileResult]
[ProfileAll]
public string FilterTest()
{
    return "This is the FilterTest action";
}
```

Вы можете увидеть эффект от всех этих фильтров, если запустите приложение и перейдите по ссылке `/Home/FilterTest`. Результат показан на рисунке 16-10.

#### Рисунок 16-10: Эффект добавления фильтра `ProfileAll`



# Использование других возможностей фильтров

В предыдущих примерах мы разобрали все, что необходимо для эффективной работы с фильтрами. Далее мы продемонстрируем вам некоторые дополнительные возможности фильтрации MVC Framework, которые представляют некоторый интерес, хотя и не так широко используются.

## Фильтрация без атрибутов

Обычно для использования фильтров мы применяем атрибуты, как это уже было показано в предыдущих разделах. Тем не менее, этому есть альтернатива - класс `Controller` реализует интерфейсы `IAuthorizationFilter`, `IActionFilter`, `IResultFilter` и `IExceptionFilter`. Он также предоставляет пустые виртуальные реализации всех методов `OnXXX`, с которыми вы уже также знакомы, например, `OnAuthorization` и `OnException`. В листинге 16-32 мы обновили контроллер `Home`, чтобы использовать эту функцию и создать класса контроллера, который будет самостоятельно анализировать свою производительность.

**Листинг 16-32:** Используем методы фильтров контроллера

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using Filters.Infrastructure;
using System.Diagnostics;
namespace Filters.Controllers
{
    public class HomeController : Controller
    {
        private Stopwatch timer;
        [Authorize(Users = "adam, steve, jacqui", Roles = "admin")]
        public string Index()
        {
            return "This is the Index action on the Home controller";
        }
        [HandleError(ExceptionType = typeof(ArgumentOutOfRangeException), View =
"RangeError")]
        public string RangeTest(int id)
        {
            if (id > 100)
            {
                return String.Format("The id value is: {0}", id);
            }
            else
            {
                throw new ArgumentOutOfRangeException("id", id, "");
            }
        }
        public string FilterTest()
        {
            return "This is the FilterTest action";
        }
        protected override void OnActionExecuting(ActionExecutingContext filterContext)
        {
            timer = Stopwatch.StartNew();
        }
        protected override void OnResultExecuted(ResultExecutedContext filterContext)
        {
            timer.Stop();
            filterContext.HttpContext.Response.Write(

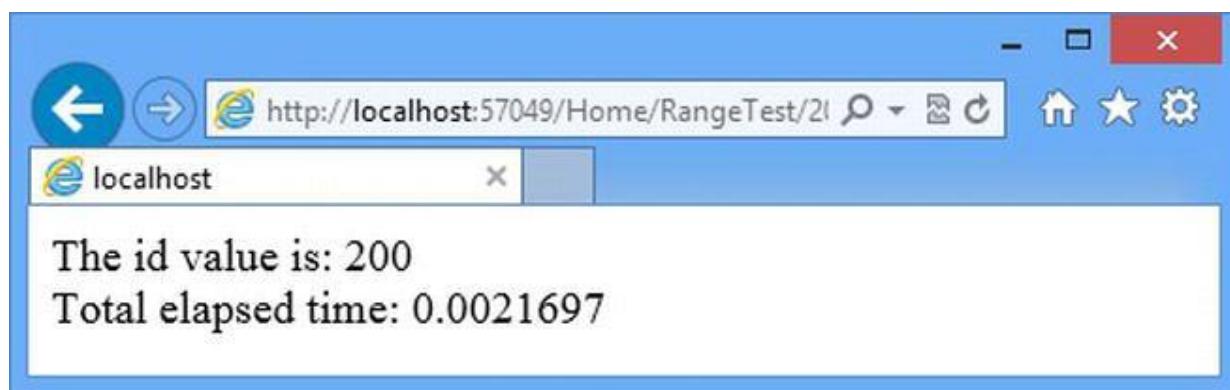
```

```
        string.Format("<div>Total elapsed time: {0}</div>",  
                      timer.Elapsed.TotalSeconds));  
    }  
}
```

Мы удалили фильтры из метода действия `FilterTest`, потому что они больше не требуется - контроллер `Home` будет добавлять информацию о производительности к ответу для каждого метода действия.

На рисунке 16-11 показан эффект запуска программы и перехода по ссылке /Home/RangeTest/200, которая приведет нас к действию RangeTest, не вызывая исключения, которое мы создали для демонстрации фильтра HandleError.

**Рисунок 16-11:** Эффект реализации методов фильтра непосредственно в контроллере



Этот подход будет полезен, когда вы создаете базовый класс, от которого наследуют несколько контроллеров вашего проекта. Весь смысл фильтрации заключается в том, чтобы заключить код, необходимый всему приложению, в один модуль, подразумевающий многоразовое использование. Таким образом, бессмыленно использовать эти методы в классе, который не будет использоваться как базовый контроллер.

## **Подсказка**

*В наших проектах мы предпочитаем использовать атрибуты: нам нравится разделять логику контроллеров и фильтров. Если вы хотите применить фильтр ко всем контроллерам, то можете использовать для этого глобальные фильтры.*

## Используем глобальные фильтры

Глобальные фильтры применяются ко всем методам действий во всех контроллерах приложения. Превратить обычный фильтр в глобальный можно с помощью метода `RegisterGlobalFilters`, определенного в файле `App_Start/FilterConfig.cs`. В листинге 16-33 показано, как сделать фильтр `ProfileAll`, который мы создали в листинге 16-30, глобальным.

**Листинг 16-33:** Создаем глобальный фильтр в файле FilterConfig.cs

```
using System.Web;
using System.Web.Mvc;
using Filters.Infrastructure;
namespace Filters
{
    public class FilterConfig
```

```

    {
        public static void RegisterGlobalFilters(GlobalFilterCollection filters)
        {
            filters.Add(new HandleErrorAttribute());
            filters.Add(new ProfileAllAttribute());
        }
    }
}

```

Метод RegisterGlobalFilters вызывается из метода Application\_Start в файле Global.asax, что гарантирует регистрацию глобальных фильтров при запуске приложения.

#### *Примечание*

*Первый оператор в методе RegisterGlobalFilters, создаваемом Visual Studio по умолчанию, настраивает стандартные правила обработки исключений MVC. Он будет визуализировать представление /Views/Shared/Error.cshtml при возникновении необработанного исключения. Во время разработки эти правила обработки исключений по умолчанию отключены. В секции "Создание фильтра исключения" далее в этой главе говорится о том, как ее включить в файле Web.config.*

---

Параметром для метода RegisterGlobalFilters является GlobalFilterCollection. Чтобы зарегистрировать глобальный фильтр, используйте метод Add, например:

```
filters.Add(new ProfileAllAttribute());
```

Обратите внимание, что к фильтру нужно обращаться, используя полное имя класса (ProfileAllAttribute), а не краткое, которое применяется при обращении к фильтру как к атрибуту (ProfileAll). Фильтр, зарегистрированный таким образом, будет применен к каждому действию метода.

Чтобы продемонстрировать работу глобальных фильтров, мы создали контроллер под названием Customer, как показано в листинге 16-34. Новый контроллер нам нужен потому, что мы хотим использовать код, к которому не применялись фильтры в предыдущих разделах.

#### **Листинг 16-34:** Контроллер Customer

```

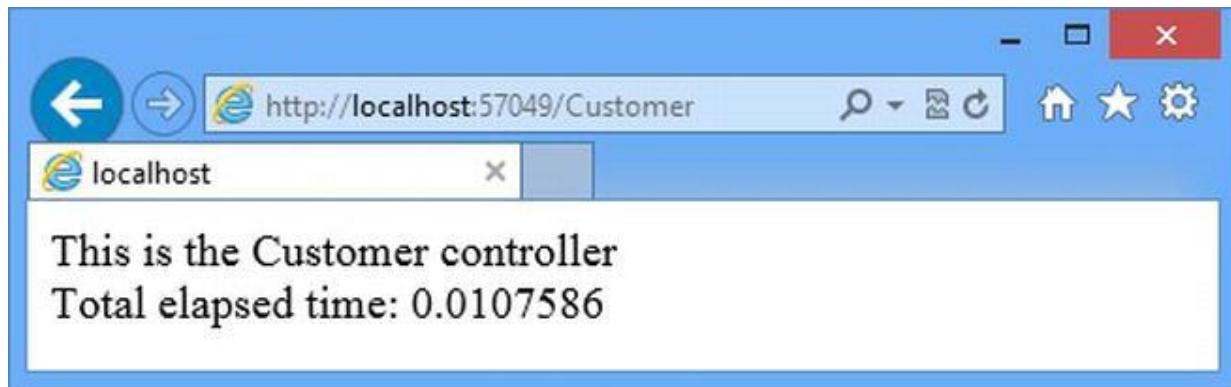
using System.Web.Mvc;
namespace Filters.Controllers
{
    public class CustomerController : Controller
    {
        public string Index()
        {
            return "This is the Customer controller";
        }
    }
}

```

Это очень простой контроллер, в котором есть действие Index, возвращающее строку. На рисунке 16-12 показан эффект применения глобального фильтра, который мы увидели, запустив приложение и перейдя по ссылке /Customer.

Хотя мы не применяли фильтр непосредственно к контроллеру, глобальный фильтр добавляет информацию о производительности, как показано на рисунке.

**Рисунок 16-12:** Эффект применения глобального фильтра



### Задаем порядок выполнения фильтров

Мы уже говорили о том, что фильтры выполняются в определенной последовательности, в зависимости от типа: фильтры авторизации, фильтры действий и затем фильтры результатов. Фильтры исключений выполняются на любом этапе, когда появляется необработанное исключение. Однако внутри типа порядок выполнения отдельных фильтров определяется вами.

В листинге 16-35 показан простой класс фильтра действий под названием `SimpleMessageAttribute`, который мы добавили в папку `Infrastructure` и на примере которого продемонстрируем, как задавать порядок выполнения фильтров.

#### Листинг 16-35: Простой фильтр действий

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
namespace Filters.Infrastructure
{
    [AttributeUsage(AttributeTargets.Method, AllowMultiple = true)]
    public class SimpleMessageAttribute : FilterAttribute, IActionFilter
    {
        public string Message { get; set; }
        public void OnActionExecuting(ActionExecutingContext filterContext)
        {
            filterContext.HttpContext.Response.Write(
                string.Format("<div>[Before Action: {0}]</div>", Message));
        }
        public void OnActionExecuted(ActionExecutedContext filterContext)
        {
            filterContext.HttpContext.Response.Write(
                string.Format("<div>[After Action: {0}]</div>", Message));
        }
    }
}
```

При вызове методов `OnActionExecuting` и `OnActionExecuted` этот фильтр записывает в ответ сообщение, часть которого задается с помощью свойства `Message`.

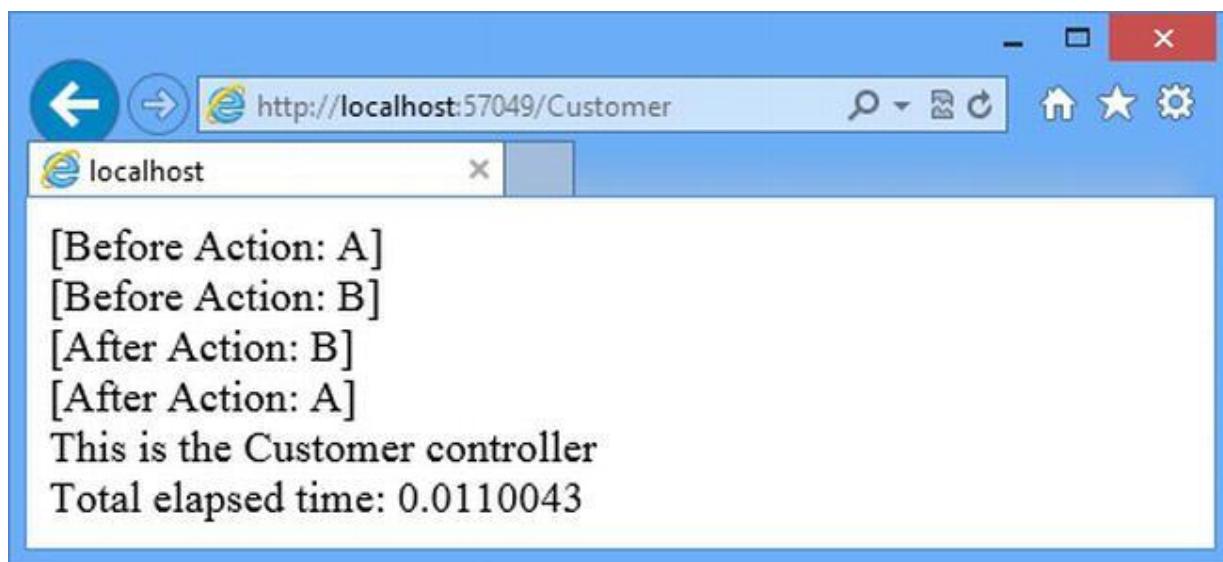
Мы можем применить несколько экземпляров этого фильтра к методу действия, как показано в листинге 16-36 ( обратите внимание, что в атрибуте `AttributeUsage` мы устанавливаем свойству `AllowMultiple` значение `true`).

### Листинг 16-36: Применяем несколько фильтров к действию

```
using System.Web.Mvc;
using Filters.Infrastructure;
namespace Filters.Controllers
{
    public class CustomerController : Controller
    {
        [SimpleMessage(Message = "A")]
        [SimpleMessage(Message = "B")]
        public string Index()
        {
            return "This is the Customer controller";
        }
    }
}
```

Мы создали два фильтра с разными сообщениями: в первом – сообщение А, во втором - сообщение В. Мы могли бы использовать два различных фильтра, но такой подход позволит упростить наш пример. Когда вы запустите приложение и перейдете по ссылке /Customer, то увидите результат, показанный на рисунке 16-13.

Рисунок 16-13: Несколько фильтров в одном методе действия



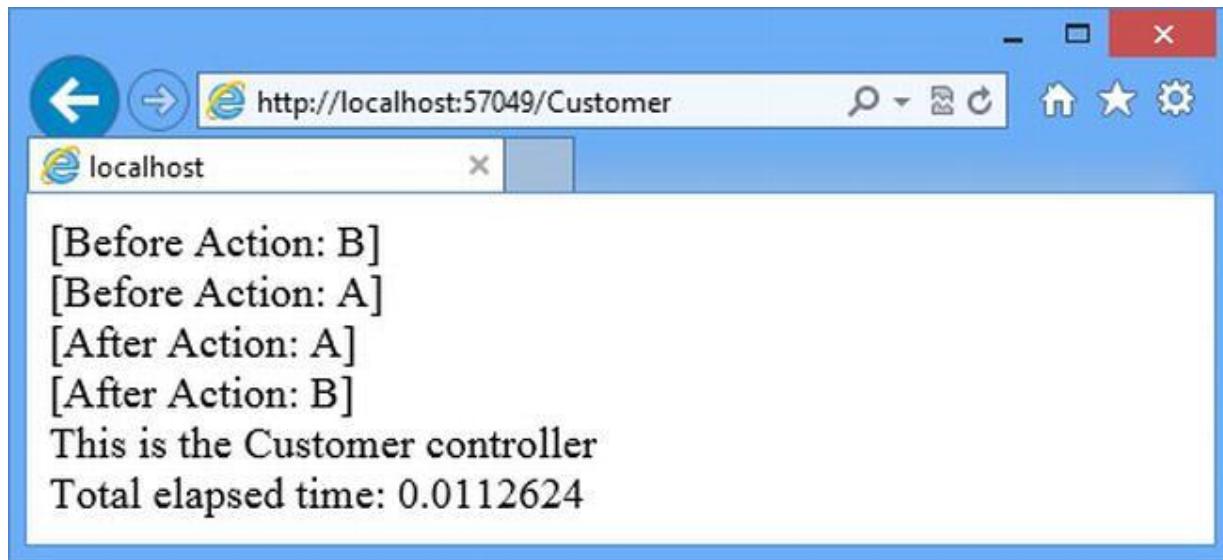
Когда мы запустим этот пример, MVC Framework выполнит фильтр А перед фильтром В, но могло быть и наоборот – платформа не гарантирует какого-либо определенного порядка выполнения. В большинстве случаев порядок не имеет значения, но если он важен, используйте свойство Order, как показано в листинге 16-37.

### Листинг 16-37: Используем свойство Order в фильтре

```
[SimpleMessage(Message = "A", Order = 2)]
[SimpleMessage(Message = "B", Order = 1)]
public ActionResult Index()
{
    Response.Write("Action method is running");
    return View();
}
```

Параметр Order принимает значение int, и MVC Framework выполняет фильтры в порядке его возрастания. В листинге мы назначили фильтру в наименьшее значение, поэтому он выполняется первым, как показано на рисунке 16-14.

**Рисунок 16-14:** Указываем порядок выполнения фильтров



#### Примечание

Обратите внимание, что методы `OnActionExecuting` выполняются в порядке, который определили мы, а методы `OnActionExecuted` - в обратном порядке. MVC Framework строит стек фильтров, когда выполняет их перед методом действия, а затем раскручивает этот стек, или возвращает его в исходное положение. Поведение для раскрутки стека нельзя изменить.

Если мы не указываем значение для свойства `Order`, по умолчанию ему присваивается значение 1. Это означает, что если вы смешиваете фильтры так, что одни имеют значения `Order`, а другие нет, то фильтры без значений будут выполняться в первую очередь, так как они имеют самые низкие значения `Order`.

Если несколько фильтров одного типа (например, фильтров действий) имеют одинаковое значение `Order` (скажем, 1), то MVC Framework определяет порядок выполнения на основании того, к чему применен фильтр. Глобальные фильтры выполняются в первую очередь, затем - фильтры, примененные к классу контроллера, затем - к методу действия.

#### Примечание

*Для фильтров исключений применяется обратный порядок исполнения. Если фильтры исключений с одинаковым значением `Order` применяются к контроллеру и методу действия, фильтр метода действия выполняется в первую очередь. Глобальный фильтр исключений с тем же значением `Order` будет выполнен последним.*

## Использование встроенных фильтров

MVC Framework предоставляет некоторые встроенные фильтры, которые мы описали в таблице 16-9.

**Таблица 16-9:** Встроенные фильтры

Фильтр	Описание
<code>RequireHttps</code>	Разрешает использование протокола HTTPS для действий

Фильтр	Описание
OutputCache	Кэширует вывод из метода действия
ValidateInput и ValidationAntiForgeryToken	Фильтры авторизации, связанные с безопасностью
AsyncTimeout и NoAsyncTimeout	Используются в асинхронных контроллерах
ChildActionOnlyAttribute	Фильтр авторизации, который поддерживает вспомогательные методы <code>Html.Action</code> и <code>Html.RenderAction</code>

Большинство этих фильтров описаны в других частях книги. Тем не менее, два фильтра - `RequireHttps` и `OutputCache` – не относятся к чему-либо конкретно, так что мы разберем их применение здесь.

## Используем фильтр `RequireHttps`

Фильтр `RequireHttps` разрешает использование протокола HTTPS для действий. Он перенаправляет браузер пользователя к тому же самому действию, но используя префикс протокола `https://`.

Вы можете переопределить метод `HandleNonHttpsRequest` и создать пользовательское поведение для обработки незащищенных запросов. Этот фильтр применяется только к запросам GET. Данные формы будут потеряны, если таким образом будет перенаправлен запрос POST.

### Примечание

*При использовании фильтра `RequireHttps` у вас могут возникнуть проблемы с порядком выполнения фильтров, потому что `RequireHttps` – фильтр авторизации, а не фильтр действия. Информация о порядке выполнения фильтров дана в разделе «Задаем порядок выполнения фильтров» ранее в этой главе.*

## Используем фильтр `OutputCache`

Фильтр `OutputCache` запускает кэширование вывода от метода действия таким образом, чтобы одно и то же содержание могло быть повторно использовано для обслуживания последующих запросов к одному URL. Кэширование вывода действия может значительно увеличить производительность, так как позволяет избежать многих отнимающих время этапов при обработке запроса (например, обращение к базе данных). Конечно, недостатком кэширования является то, что вы сможете предоставлять только один ответ на все запросы, что подходит не для всех методов действий.

Фильтр `OutputCache` использует механизм кэширования вывода из ядра платформы ASP.NET, и если вы когда-либо использовали кэширование в приложениях Web Forms, то параметры конфигурации будут вам знакомы. Фильтр `OutputCache` можно использовать для контроля кэширования на стороне клиента, изменения значения, отправляемые в заголовке `Cache-Control`. В таблице 16-10 показаны параметры, которые можно установить для этого фильтра.

**Таблица 16-10:** Параметры для фильтра `OutputCache`

Параметр	Тип	Описание
Duration	int	Обязательный – определяет, как долго вывод остается в кэше (в секундах).
VaryByParam	string (список, разделенный точкой с	Сообщает ASP.NET использовать новую запись кэша для каждой комбинации значений

Параметр	Тип	Описание
	запятой)	Request.QueryString и Request.Form, совпадающей с указанными именами. Значение по умолчанию, none, означает «не изменять по значениям строки запроса или формы». Другой вариант, *, означает «изменять по значениям строки запроса или формы». Если не указано иное, используется значение none.
VaryByHeader	string (список, разделенный точкой с запятой)	Сообщает ASP.NET использовать новую запись кэша для каждой комбинации значений, отправленных в названиях заголовков HTTP.
VaryByCustom	string	Если указано, ASP.NET вызывает метод GetVaryByCustomString в файле Global.asax, передает это произвольное значение строки в качестве параметра, что позволит вам генерировать свой собственный ключ кэша. Специальное значение browser позволяет изменять кэш по названию и старшему номеру версии браузера.
VaryByContentEncoding	string (список, разделенный точкой с запятой)	Позволяет ASP.NET создать отдельный кэш для каждого закодированного содержимого (например, gzip и deflate), которое может быть запрошено браузером.
Location	OutputCacheLocation	Указывает, где должен быть закэширован вывод. Он принимает одно из следующих значений : Server (только в памяти сервера), Client (только в браузере пользователя), Downstream (в браузере пользователя или любом промежуточном устройстве, поддерживающем кэширование HTTP, таком как прокси-сервер), ServerAndClient (объединение Server и Client), Any (объединение Server и Downstream) либо None (кэширование не используется). Если не указано иное, параметр принимает значение по умолчанию Any.
NoStore	bool	Если содержит true, то ASP.NET отправит заголовок Cache-Control: no-store в браузер, что укажет браузеру кэшировать страницу только в течение того времени, которое необходимо для ее отображения. Параметр используется только для защиты очень чувствительных данных.
CacheProfile	string	Параметр указывает ASP.NET получить параметры кэша из раздела под названием <outputCacheSettings> в файле Web.config.
SqlDependency	string	Если указать пару база данных/имя, данные в кэше будут удалены автоматически при изменении соответствующих данных из базы. Для этого требуется функция SQL cache dependency, которую довольно сложно настроить. Более подробно о ней можно почитать на <a href="http://msdn.microsoft.com/en-us/library/ms178604.aspx">http://msdn.microsoft.com/en-us/library/ms178604.aspx</a> .

Одной из сильных сторон фильтра `OutputCache` является то, что его можно применять к дочерним действиям. Дочернее действие вызывается из представления с помощью вспомогательного метода `Html.Action`. Это позволяет нам выбирать, какие части ответа кэшируются и какие генерируются динамически. Мы обсудим дочерние действия подробно в главе 18, но в листинге 16-38 продемонстрируем работу с ними на примере простого контроллера под названием `SelectiveCache`.

#### Листинг 16-38: Контроллер `SelectiveCache`

```
using System;
using System.Web.Mvc;
namespace Filters.Controllers
{
    public class SelectiveCacheController : Controller
    {
        public ActionResult Index()
        {
            Response.Write("Action method is running: " + DateTime.Now);
            return View();
        }
        [OutputCache(Duration = 30)]
        public ActionResult ChildAction()
        {
            Response.Write("Child action method is running: " + DateTime.Now);
            return View();
        }
    }
}
```

Контроллер в листинге 16-38 определяет два метода действия:

- Метод `ChildAction`, к которому применен фильтр `OutputCache`. Этот метод действия мы будем вызывать из представления.
- Метод действия `Index`, который будет родительским действием.

Оба метода действий записывают время, которое потребовалось для их выполнения, в объект `Response`. В листинге 16-39 показано представление `Index.cshtml` (связанное с методом действия `Index`).

#### Листинг 16-39: Представление, который вызывает кэшированное дочернее действие

```
@{
    ViewBag.Title = "Index";
}
<h2>This is the main action view</h2>
@Html.Action("ChildAction")
```

Как видите, в конце представления мы вызываем метод `ChildAction`. Представление для этого метода показано в листинге 16-40.

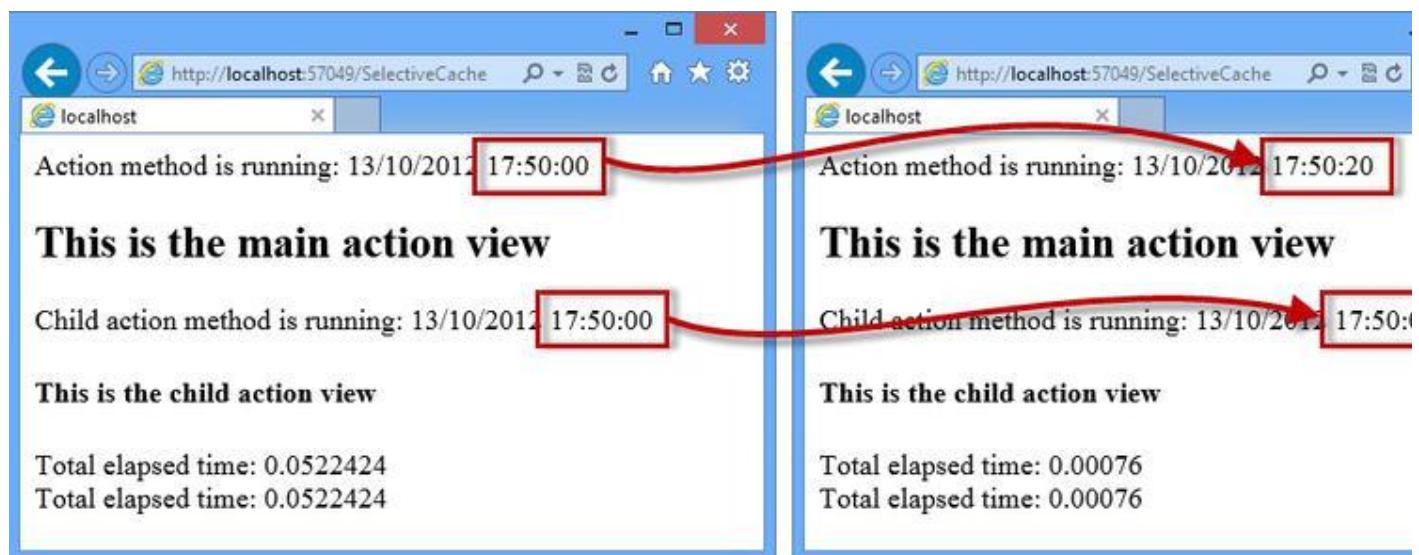
#### Листинг 16-40: Представление `ChildAction.cshtml`

```
@{
    Layout = null;
}
<h4>This is the child action view</h4>
```

Запустите приложение и перейдите по ссылке `/selectiveCache`. В первый раз вы увидите, что и родительское, и дочернее действие указывают в своих ответах одно и то же время. Если вы перезагрузите страницу (или перейдете по той же ссылке в другом браузере), то увидите, что время, указанное родительским действием, изменилось, но время дочернего действия осталось прежним.

Это говорит нам о том, что мы видим кэшированный вывод от первоначального вызова, как показано на рисунке 16-15.

**Рисунок 16-15:** Эффект кэширования вывода дочернего действия



#### Подсказка

*Возможно, вам придется обновить страницу еще раз, прежде чем начнется кэширование - это объясняется способом компиляции представлений при первом запуске приложения MVC (который мы рассмотрим в главе 18).*

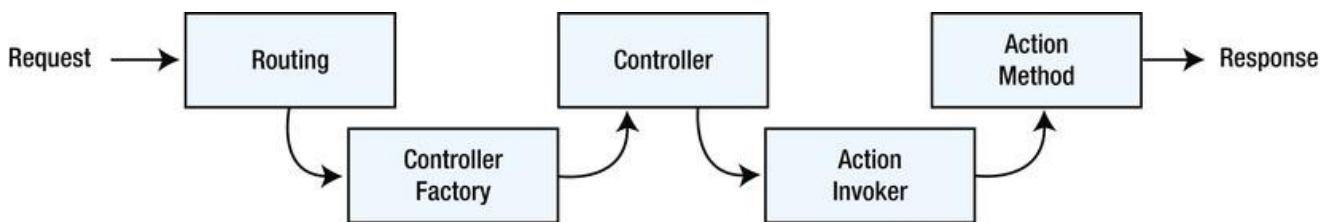
## Резюме

В этой главе мы рассмотрели фильтры как способ инкапсуляции логики, которая относится к сквозной функциональности. Мы изучили различные виды фильтров и их реализации. Мы научились применять фильтры в качестве атрибутов к контроллерам и методам действий, а так же в качестве глобальных фильтров. Фильтры расширяют логику обработки запроса, избавляя нас от необходимости включать дополнительную логику в методы действий. В следующей главе мы научимся изменять и расширять поведение MVC Framework для работы с контроллерами.

# Расширенные возможности контроллера

В этой главе мы покажем вам некоторые дополнительные возможности MVC для работы с контроллерами. Для начала рассмотрим элементы конвейера обработки запросов, которые запускают выполнение метода действия, и продемонстрируем различные способы управления этим процессом. На рисунке 17-1 показан базовый поток передачи управления между компонентами.

Рисунок 17-1: Вызов метода действия



В первой части этой главы в центре нашего внимания будет фабрика контроллеров и средство вызова действий. Назначение этих компонентов можно понять по их названиям. Фабрика контроллеров создает экземпляры контроллеров для обслуживания запроса, а средство вызова действий отвечает за поиск и вызов методов действий в классе контроллера. MVC Framework включает стандартные реализации обоих компонентов, и мы рассмотрим, как их конфигурировать и контролировать их поведение. Мы также покажем, как заменять эти компоненты и использовать пользовательскую логику.

## Создание проекта для примера

Для этой главы мы создадим новый проект MVC под названием `ControllerExtensibility` на шаблоне `Empty`. Для работы нам потребуются несколько простых контроллеров, чтобы продемонстрировать различные возможности расширения. Для начала определим класс `Result.cs` в папке `Models`, содержимое которого показано в листинге 17-1.

Листинг 17-1: Объект модели `Result`

```
namespace ControllerExtensibility.Models
{
    public class Result
    {
        public string ControllerName { get; set; }
        public string ActionName { get; set; }
    }
}
```

Далее мы создадим папку `/Views/Shared` и добавим новое представление под названием `Result.cshtml`. Это представление, которое будут визуализировать все методы действий в наших классах контроллерах, его содержимое показано в листинге 17-2.

Листинг 17-2: Содержимое файла `Result.cshtml`

```
@model ControllerExtensibility.Models.Result

@{
    Layout = null;
}

<!DOCTYPE html>
<html>
<head>
```

```

<meta name="viewport" content="width=device-width" />
<title>Result</title>
</head>
<body>
<div>Controller: @Model.ControllerName</div>
<div>Action: @Model.ActionName</div>
</body>
</html>

```

Это представление использует класс `Result`, который мы определили в листинге 17-1, в качестве модели и просто отображает значения свойств `ControllerName` и `ActionName`. Наконец, нам нужно создать несколько базовых контроллеров.

В листинге 17-3 показан контроллер `Product`.

#### **Листинг 17-3:** Контроллер `Product`

```

using ControllerExtensibility.Models;
using System.Web.Mvc;

namespace ControllerExtensibility.Controllers
{
    public class ProductController : Controller
    {
        public ViewResult Index()
        {
            return View("Result", new Result
            {
                ControllerName = "Product",
                ActionName = "Index"
            });
        }

        public ViewResult List()
        {
            return View("Result", new Result
            {
                ControllerName = "Product",
                ActionName = "List"
            });
        }
    }
}

```

В листинге 17-4 показан контроллер `Customer`.

#### **Листинг 17-4:** Контроллер `Customer`

```

using ControllerExtensibility.Models;
using System.Web.Mvc;

namespace ControllerExtensibility.Controllers
{
    public class CustomerController : Controller
    {
        public ViewResult Index()
        {
            return View("Result", new Result
            {
                ControllerName = "Customer",
                ActionName = "Index"
            });
        }
    }
}

```

```

public ViewResult List()
{
    return View("Result", new Result
    {
        ControllerName = "Customer",
        ActionName = "List"
    });
}
}
}

```

Наши контроллеры сообщают представлению `Result.cshtml`, что было вызвано, – больше ничего полезного они не делают. В этой главе нам больше ничего и не нужно, так как здесь мы учимся изменять способы обработки контроллеров и действий.

## Создание пользовательской "фабрики" контроллеров (controller factory)

Как и для большинства других элементов MVC Framework, лучший способ понять принцип работы фабрики контроллеров – это создать ее пользовательскую реализацию. Мы не рекомендуем вам делать это в реальных проектах, так как гораздо проще создать пользовательское поведение, расширяя встроенную фабрику. Таким образом, мы хотим только продемонстрировать, как MVC Framework создает экземпляры контроллеров. Фабрики контроллеров определяются интерфейсом `IControllerFactory`, который показан в листинге 17-5.

### Листинг 17-5: Интерфейс `IControllerFactory`

```

using System.Web.Routing;
using System.Web.SessionState;
namespace System.Web.Mvc
{
    public interface IControllerFactory
    {
        IController CreateController(RequestContext requestContext, string
controllerName);
        SessionStateBehavior GetControllerSessionBehavior(RequestContext requestContext,
string controllerName);
        void ReleaseController(IController controller);
    }
}

```

В следующих разделах мы создадим простую пользовательскую фабрику контроллеров и продемонстрируем реализации для каждого метода в интерфейсе `IControllerFactory`. Для этого добавим папку `Infrastructure` и в ней определим класс `CustomControllerFactory.cs`. Пользовательская фабрика контроллеров показана в листинге 17-6.

### Листинг 17-6: Содержимое файла `CustomControllerFactory.cs`

```

using System;
using System.Web.Mvc;
using System.Web.Routing;
using System.Web.SessionState;
using ControllerExtensibility.Controllers;

namespace ControllerExtensibility.Infrastructure
{
    public class CustomControllerFactory : IControllerFactory

```

```

{
    public IController CreateController(RequestContext requestContext, string
controllerName)
    {
        Type targetType = null;
        switch (controllerName)
        {
            case "Product":
                targetType = typeof(ProductController);
                break;
            case "Customer":
                targetType = typeof(CustomerController);
                break;
            default:
                requestContext.RouteData.Values["controller"] = "Product";
                targetType = typeof(ProductController);
                break;
        }
        return targetType == null ? null :
            (IController)DependencyResolver.Current.GetService(targetType);
    }

    public SessionStateBehavior GetControllerSessionBehavior(RequestContext
requestContext,
        string controllerName)
    {
        return SessionStateBehavior.Default;
    }

    public void ReleaseController(IController controller)
    {
        IDisposable disposable = controller as IDisposable;
        if (disposable != null)
        {
            disposable.Dispose();
        }
    }
}
}

```

Самый важный метод в интерфейсе - это `CreateController`, который вызывается, когда платформе требуется контроллер для обслуживания запроса. Параметрами этого метода являются объект `RequestContext`, который позволяет фабрике просматривать информацию о запросе, и `string`, который содержит имя контроллера, полученное из URL. Класс `RequestContext` определяет свойства, описанные в таблице 17-1.

**Таблица 17-1:** Свойства `RequestContext`

Название	Тип	Описание
HttpContext	HttpContextBase	Предоставляет информацию о HTTP-запросе
RouteData	RouteData	Предоставляет информацию о маршруте, который соответствует запросу

Мы не рекомендуем создавать пользовательский контроллер таким образом в том числе и потому, что поиск классов контроллеров в приложении и создание их экземпляров – сложный процесс, вы должны быть в состоянии динамически и последовательно размещать контроллеры, а также устранить все виды потенциальных проблем, такие как различие классов с одинаковыми именами в разных пространствах имен, исключения конструкторов и многие другие.

В нашем примере проекта только два контроллера, и для создания их экземпляров мы пропишем имена соответствующих классов в фабрике контроллеров – определенно, это не самая лучшая идея для реального проекта.

Цель метода `CreateController` – создание экземпляров классов контроллеров, которые могут обработать текущий запрос. Для него нет никаких ограничений, только одно единственное правило, согласно которому результатом метода должен быть объект, который реализует интерфейс `IController`.

Соглашения, которые вы уже встречали ранее в этой книге, существуют для того, чтобы пользовательская фабрика контроллеров была написана по образцу стандартной. К примеру, мы реализовали одно из таких соглашений в коде: когда мы получаем запрос к контроллеру, мы добавляем к его имени `Controller`, так что запрос к `Product` приводит к созданию экземпляра класса `ProductController`.

Создавая фабрику контроллеров, вы можете следовать соглашениям MVC или отказаться от них и создать свои собственные, которые больше подходят вашему проекту. Мы не думаем, что имеет смысл создавать новые соглашения только ради создания новых соглашений, но это может проиллюстрировать гибкость платформы MVC.

## Резервный контроллер

Пользовательская фабрика контроллеров должна возвращать реализацию интерфейса `IController` как результат метода `CreateController`, иначе будет отображена ошибка. Это означает, что нужно иметь запасной вариант для запросов, которым не будет соответствовать ни один контроллер в вашем проекте. Эту проблему можно решать по-разному: вы можете определить специальный контроллер, который будет отображать сообщение об ошибке, или, что и сделали мы, соотнести запрос с каким-нибудь другим классом контроллера.

Когда мы получаем запрос, который не соответствует ни одному контроллеру в нашем проекте, мы соотносим его с классом `ProductController`. Это может быть не самым полезным решением в реальном проекте, но здесь оно демонстрирует, что фабрика контроллеров обладает полной гибкостью в интерпретации запросов. Однако здесь вам необходимо понимать, как работают другие компоненты MVC Framework.

По умолчанию MVC выбирает представление на основе значения `controller` в данных маршрутизации, а не имени класса контроллера. Итак, если мы хотим, чтобы запасной контроллер в нашем примере работал с представлениями, которые организованы по имени контроллера, нам необходимо изменить свойство маршрутизации `controller`:

```
requestContext.RouteData.Values["controller"] = "Product";
```

Это изменение приведет к тому, MVC Framework будет искать представления, связанные с нашим резервным контроллером, а не тем, который был запрошен пользователем.

Здесь есть два важных момента. Во-первых, фабрика контроллеров не только соотносит запросы с контроллерами, но и может изменять запросы, чтобы повлиять на поведение на определенных этапах в конвейере обработки запроса. У этой ее возможности большой потенциал, и она является важной характеристикой MVC Framework.

Во-вторых, даже если вы можете создавать любые соглашения для фабрики контроллеров, не забывайте, что соглашения соблюдаются и в других компонентах MVC Framework. И, так как эти другие компоненты также могут быть заменены на пользовательский код (например, как

представления в главе 18), то имеет смысл следовать соглашениям везде, где это возможно, благодаря чему компоненты можно будет разрабатывать и использовать независимо друг от друга.

## Создаем экземпляры классов контроллеров

Создание экземпляров классов контроллеров не регулируется никакими правилами, но удобнее всего использовать для этого преобразователь зависимостей (DR), который мы рассматривали в главе 6. Он позволит сохранить главной задачей пользовательской фабрики сопоставление запросов с классами контроллеров, а такие вопросы, как внедрение зависимостей, будут обрабатываться отдельно и для всего приложения. Для создания экземпляров наших контроллеров мы использовали класс DependencyResolver:

```
return targetType == null ? null :  
    (IController) DependencyResolver.Current.GetService(targetType);
```

Статическое свойство DependencyResolver.Current возвращает реализацию интерфейса IDependencyResolver, который определяет метод GetService. Вы передаете объект System.Type в этот метод, а он возвращает экземпляр этого объекта. Существует строго типизированный вариант метода GetService, но так как заранее мы не знаем, с каким типом будем работать, то придется использовать вариант, который возвращает Object, а затем явно передает его в IController.

### Подсказка

*Мы не проверяли, является ли объект, который возвращает метод GetService, реализацией IController, но в реальных проектах это нужно делать, особенно если вы используете классы, созданные другими разработчиками.*

## Реализуем другие методы интерфейса

В интерфейсе IControllerFactory осталось два метода:

- Метод GetControllerSessionBehavior используется MVC Framework, чтобы определить, нужно ли предоставлять контроллеру данные сессии. К нему мы вернемся в разделе "Используем контроллеры без поддержки состояния сессии" далее в этой главе.
- Метод ReleaseController вызывается, когда объект контроллера, созданный методом CreateController, больше не нужен. В нашей реализации мы проверяем, реализует ли класс интерфейс IDisposable. Если реализует, мы вызываем метод Dispose, чтобы освободить все ресурсы, которые возможно.

Наша реализация методов GetControllerSessionBehaviour и ReleaseController подходит для большинства проектов и ее можно использовать без изменений (хотя сначала прочитайте раздел о контроллерах без поддержки состояния сессии далее в этой главе, чтобы убедиться, что вы понимаете все доступные варианты).

## Регистрируем пользовательскую фабрику контроллеров

Чтобы сообщить MVC Framework, что он должен использовать пользовательскую фабрику контроллеров, мы используем класс ControllerBuilder. Зарегистрировать пользовательскую фабрику нужно при запуске приложения, то есть с помощью метода Application\_Start в файле Global.asax.cs, как показано в листинге 17-7.

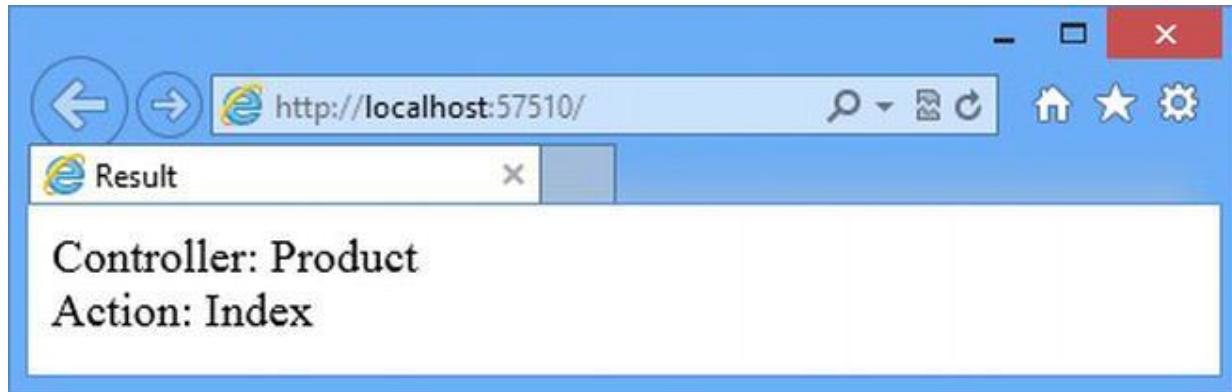
### Листинг 17-7: Регистрируем пользовательскую фабрику контроллеров

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Http;
using System.Web.Mvc;
using System.Web.Routing;
using ControllerExtensibility.Infrastructure;

namespace ControllerExtensibility
{
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();
            WebApiConfig.Register(GlobalConfiguration.Configuration);
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            ControllerBuilder.Current.SetControllerFactory(new CustomControllerFactory());
        }
    }
}
```

Когда фабрика зарегистрирована, она будет обрабатывать все запросы, которые поступают в приложение. Вы можете увидеть эффект применения пользовательской фабрики, просто запустив приложение - браузер запросит корневой URL, который будет отображен системой маршрутизации в контроллер `Home`. Наша пользовательская фабрика обработает запрос для контроллера `Home`, создав экземпляра класса `ProductController`, что приведет к результату, показанному на рисунке 17-2.

Рисунок 17-2: Используем пользовательскую фабрику контроллеров



## Работа со встроенной фабрикой контроллеров

Мы создали пользовательскую фабрику контроллеров, потому что это наиболее эффективный способ показать, какими обязанностями обладает фабрика контроллеров и как она функционирует. Однако в большинстве случаев встроенный класс фабрики контроллеров, `DefaultControllerFactory`, будет нас вполне удовлетворять. Получая запрос от системы маршрутизации, эта фабрика просматривает данные маршрутизации, чтобы найти значение свойства `controller` (которое мы изучили в главе 13), и пытается найти в приложении класс, который отвечает следующим критериям:

- класс должен быть помечен как `public`;
- класс должен быть конкретным (не помечен как `abstract`);

- класс не должен принимать общие (*generic*) параметры;
- имя класса должно заканчиваться на `Controller`;
- класс должен реализовывать интерфейс `IController`.

Класс `DefaultControllerFactory` поддерживает список таких классов в приложении, так что ему не нужно выполнять поиск каждый раз, когда поступает запрос. Если подходящий класс существует, то он создает экземпляр с помощью активатора контроллера (мы вернемся к этому в следующем разделе "Настраиваем процедуру создания экземпляров контроллеров в `DefaultControllerFactory`"), и на этом его работа завершена. Если подходящего контроллера нет, то запрос не может быть обработан далее.

Обратите внимание на то, что для класса `DefaultControllerFactory` соглашение имеет высший приоритет, чем конфигурация (*convention over configuration pattern*). Вам не нужно регистрировать контроллеры в файле конфигурации, потому что их найдет для вас фабрика. Все, что от вас потребуется - это создать классы, отвечающие критериям, по которым их ищет фабрика.

Если вы хотите создать пользовательское поведение фабрики контроллеров, можно изменить настройки стандартной фабрики или переопределить некоторые методы. Таким образом, вы сможете использовать поведение с высшим приоритетом соглашений, не создавая их заново и таким образом избегая наиболее сложной работы. В следующих разделах мы продемонстрируем вам различные способы изменения процедуры создания контроллеров.

## Назначаем приоритет пространствам имен

В главе 14 мы показали вам, как назначить приоритет одному или нескольким пространствам имен при создании маршрута. Это решало проблему неоднозначности контроллеров, когда классы контроллеров имели одинаковые имена, но располагались в разных пространствах имен. За обработку списка пространств имен и назначение им приоритетов отвечает `DefaultControllerFactory`.

### *Подсказка*

*Глобальные приоритеты переопределяются конкретными приоритетами, назначенными для маршрутов. Это означает, что вы можете определить глобальные правила, а затем изменить конкретные маршруты по мере необходимости. Подробнее о пространствах имен для конкретных маршрутов можно узнать в главе 14.*

---

Если у вас в приложении много маршрутов, то более удобно задавать приоритетные пространства имен глобально, чтобы они применялись ко всем маршрутам. В листинге 17-8 показано, как сделать это в методе `Application_Start` файла `Global.asax` (это наш выбор, но вы также можете использовать файл `RouteConfig.cs` в папке `App_Start`).

### Листинг 17-8: Назначаем глобальные приоритеты пространствам имен

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Http;
using System.Web.Mvc;
using System.Web.Routing;
using ControllerExtensibility.Infrastructure;
```

```

namespace ControllerExtensibility
{
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();
            WebApiConfig.Register(GlobalConfiguration.Configuration);
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);

            ControllerBuilder.Current.DefaultNamespaces.Add("MyControllerNamespace");
            ControllerBuilder.Current.DefaultNamespaces.Add("MyProject.*");
        }
    }
}

```

Мы используем статический метод `ControllerBuilder.Current.DefaultNamespaces.Add`, чтобы добавить пространства имен, которым хотим назначить приоритет. Порядок, в котором мы добавляем пространства имен, не влияет на порядок поиска и не предполагает относительный приоритет относительно друг друга - все пространства имен, определенные методом `Add`, обрабатываются одинаково, а их приоритет является относительным только по отношению к тем пространствам, которые не были добавлены методом `Add`. Это означает, что если фабрика не найдет подходящий класс контроллера в пространствах имен, определенных методом `Add`, то она произведет поиск по всему приложению.

#### *Подсказка*

*Обратите внимание, что во втором операторе, выделенным жирным шрифтом в листинге 17-8, используется символ звездочки (\*). Таким образом мы указываем, что фабрика контроллеров должна производить поиск в пространстве имен MyProject и во всех дочерних пространствах имен, которые содержит MyProject. Хотя \* и напоминает синтаксис регулярных выражений, но не является им; вы можете закончить пространства имен символом \*, но не сможете использовать никакие другие элементы синтаксиса регулярных выражений в методе Add.*

---

## Настраиваем процедуру создания экземпляров контроллеров в DefaultControllerFactory

Существует несколько способов изменить то, как класс `DefaultControllerFactory` создает экземпляры объектов контроллеров. Сейчас фабрику контроллеров наиболее часто изменяют для того, чтобы добавить поддержку DI. Это можно сделать с помощью разных приемов. Выбор наиболее подходящего будет зависеть от того, как вы используете DI в других компонентах приложения.

### Используем DR

Класс `DefaultControllerFactory` будет использовать доступный DR для создания контроллеров. Мы рассмотрели DR в главе 6 и продемонстрировали вам класс `NinjectDependencyResolver`, который реализует интерфейс `IDependencyResolver` для поддержки `Ninject DI`. Мы также использовали класс `DependencyResolver` ранее в этой главе, когда создавали пользовательскую фабрику контроллеров.

`DefaultControllerFactory` вызовет метод `IDependencyResolver.GetService`, чтобы запросить экземпляр контроллера, который дает возможность преобразовать и внедрить любую зависимость.

## Используем активатор контроллеров

Вы также можете использовать в контроллерах DI, создав активатор контроллеров (*controller activator*). Он создается реализацией интерфейса `IControllerActivator`, который показан в листинге 17-9.

### Листинг 17-9: Интерфейс `IControllerActivator`

```
namespace System.Web.Mvc
{
    using System.Web.Routing;
    public interface IControllerActivator
    {
        IController Create(RequestContext requestContext, Type controllerType);
    }
}
```

Интерфейс содержит один метод, `Create`, в который передается объект `RequestContext`, описывающий запрос, и `Type`, который определяет, для какого класса должен быть создан экземпляр. Реализация этого интерфейса показана в листинге 17-10.

### Листинг 17-10: Реализация интерфейса `IControllerActivator`

```
using ControllerExtensibility.Controllers;
using System;
using System.Web.Mvc;
using System.Web.Routing;
namespace ControllerExtensibility.Infrastructure
{
    public class CustomControllerActivator : IControllerActivator
    {
        public IController Create(RequestContext requestContext, Type controllerType)
        {
            if (controllerType == typeof(ProductController))
            {
                controllerType = typeof(CustomerController);
            }
            return (IController)DependencyResolver.Current.GetService(controllerType);
        }
    }
}
```

Реализация `IControllerActivator` довольно проста - если запрашивается класс `ProductController`, она отвечает экземпляром класса `CustomerController`. Для реального проекта это не лучшее решение, но здесь мы только показываем, как можно использовать интерфейс `IControllerActivator` для перехвата запросов между фабрикой контроллеров и DR.

Чтобы использовать пользовательский активатор, нам нужно передать экземпляр реализованного класса в конструктор `DefaultControllerFactory` и зарегистрировать результат в методе `Application_Start` файла `Global.asax`, как показано в листинге 17-11.

### Листинг 17-11: Регистрируем пользовательский активатор

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
```

```

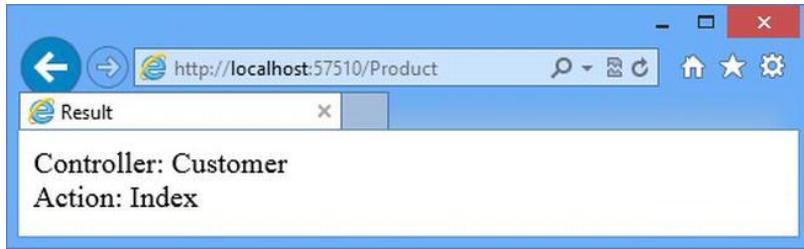
using System.Web.Http;
using System.Web.Mvc;
using System.Web.Routing;
using ControllerExtensibility.Infrastructure;
namespace ControllerExtensibility
{
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();
            WebApiConfig.Register(GlobalConfiguration.Configuration);
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);

            ControllerBuilder.Current.SetControllerFactory(new
                DefaultControllerFactory(new CustomControllerActivator()));
        }
    }
}

```

Вы можете увидеть эффект применения пользовательского активатора, если запустите приложение и перейдите по ссылке /Product. Маршрут указывает на контроллер Product, и DefaultControllerFactory попросит активатор создать экземпляра класса ProductFactory - но вместо этого активатор перехватывает запрос и создает экземпляр класса CustomerController. Результат показан на рисунке 17-3.

**Рисунок 17-3:** Перехват запросов на создание экземпляра с помощью пользовательского активатора контроллера



### Переопределяем методы DefaultControllerFactory

Чтобы изменить процедуру создания контроллеров, можно переопределить методы в классе DefaultControllerFactory. В таблице 17-2 описаны три переопределяемых метода, каждый из которых выполняет несколько специфическую задачу.

**Таблица 17-2:** Переопределяемые методы DefaultControllerFactory

Метод	Результат	Описание
CreateController	IController	Реализация метода CreateController из интерфейса IControllerFactory. По умолчанию этот метод вызывает GetControllerType, чтобы определить, для какого типа нужно создать экземпляр, а затем получает объект контроллера, передавая результат в метод GetControllerInstance.
GetControllerType	Type	Соотносит запросы с типами контроллеров. Здесь применяется большинство критериев, перечисленных ранее в этой главе.
GetControllerInstance	IController	Создает экземпляр указанного типа.

# Создание пользовательского средства вызова метода действия (action invoker)

Когда фабрика контроллеров создала экземпляр класса, платформе нужно вызвать действие в этом экземпляре. Если ваши контроллеры наследуют от класса `Controller`, то этим будет заниматься механизм вызова действий, который и является темой этого раздела.

## *Подсказка*

*Если вы создаете контроллер непосредственно из интерфейса `IController`, то вы сами должны будете позаботиться о выполнении действий. В главе 15 дана подробная информация об обоих подходах к созданию контроллеров. Механизмы вызова действий являются частью функциональности, которая включена в класс `Controller`.*

Механизм вызова действий реализует интерфейс `IActionInvoker`, который показан в листинге 17-12.

## Листинг 17-12: Интерфейс `IActionInvoker`

```
namespace System.Web.Mvc
{
    public interface IActionInvoker
    {
        bool InvokeAction(ControllerContext controllerContext, string actionPerformed);
    }
}
```

В интерфейсе есть только один член: `InvokeAction`. Параметрами являются объект `ControllerContext` (о котором мы рассказывали в главе 15) и `string`, который содержит имя вызываемого действия. Он возвращает логическое значение: `true` означает, что действие был найдено и вызвано, а `false` - что в контроллере нет соответствующего действия.

Обратите внимание, что в этом описании мы не использовали слово метод. Связь между действиями и методами является чисто опциональной. Хотя описанному выше подходу следует встроенный механизм вызова действий, вы можете обрабатывать действия любым способом, который выберете. В листинге 17-13 показана реализация интерфейса `IActionInvoker`, которая использует другой подход.

## Листинг 17-13: Пользовательский механизм вызова действий

```
using System.Web.Mvc;
namespace ControllerExtensibility.Infrastructure
{
    public class CustomActionInvoker : IActionInvoker
    {
        public bool InvokeAction(ControllerContext controllerContext, string actionPerformed)
        {
            if (actionName == "Index")
            {
                controllerContext.HttpContext.
                    Response.Write("This is output from the Index action");
                return true;
            }
            else
            {
```

```
        return false;
    }
}
```

Этот механизм вызова действий не затрагивает методы в классе контроллера. Фактически он работает с самими действиями. Если поступает запрос к действию `Index`, то он запишет сообщение непосредственно в `Response`. Если поступит запрос к любому другому действию, то он вернет `false`, после чего пользователю будет показана ошибка `404 – Not found`.

Связь механизма вызова действий с контроллером устанавливается с помощью свойства `Controller.ActionInvoker`. Это означает, что разные контроллеры в одном приложении могут использовать разные механизмы вызова. Чтобы это продемонстрировать, мы добавили в проект новый контроллер под названием `ActionInvoker`, определение которого вы можете увидеть в листинге 17-14.

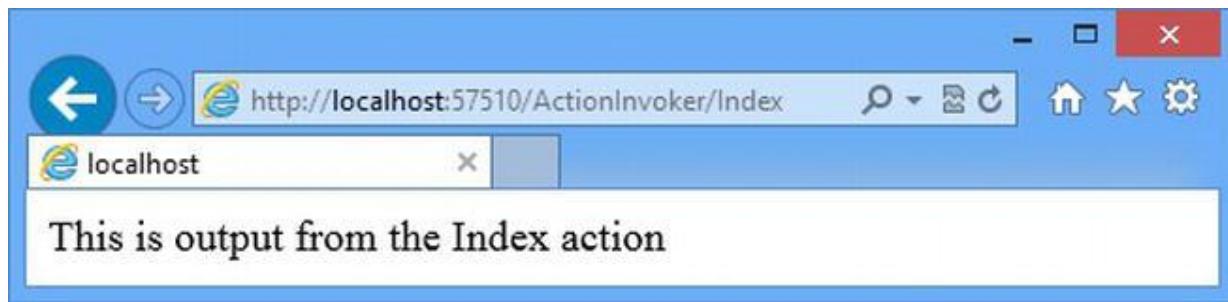
**Листинг 17-14:** Используем пользовательский механизм вызова действий в контроллере

```
using ControllerExtensibility.Infrastructure;
using System.Web.Mvc;
namespace ControllerExtensibility.Controllers
{
    public class ActionInvokerController : Controller
    {
        public ActionInvokerController()
        {
            this.ActionInvoker = new CustomActionInvoker();
        }
    }
}
```

В этом контроллере нет методов действий, и запросы обрабатывает механизм вызова действий. Вы можете увидеть, как он работает, запустив приложение и перейдя по ссылке [/ActionInvoker/Index](#).

Пользовательский механизм вызова будет генерировать ответ, показанный на рисунке 17-4. Если перейдете по ссылке, ведущей к какому-либо другому действию этого контроллера, вы увидите страницу с ошибкой 404.

**Рисунок 17-4:** Эффект применения пользовательского механизма вызова действий



Мы не думаем, что вы будете создавать собственные механизмы вызова действий, но если вы все же захотите это сделать, мы не советуем использовать этот подход. Почему? Во-первых, во встроенной поддержке есть некоторые очень полезные функции, в чем вы убедитесь в ближайшее время. Во-вторых, в нашем примере есть проблемы: плохая расширяемость, недостаточное разделение обязанностей и отсутствие поддержки представлений. Пример только демонстрирует, как работает MVC Framework и еще раз доказывает, что почти каждый элемент конвейера обработки запроса можно изменить или заменить полностью.

# Использование встроенного средства вызова метода действия

Встроенный механизм вызова действий, то есть класс `ControllerActionInvoker`, использует очень сложные технологии для сопоставления запросов с действиями. И, в отличие от нашей реализации в предыдущем разделе, механизм вызова действий по умолчанию работает с методами.

Чтобы обрабатываться как действие, метод должен соответствовать следующим критериям:

- метод должен содержать пометку `public`;
- метод *не* должен содержать пометку `static`;
- метод *не* должен присутствовать в `System.Web.Mvc.Controller` или в любом из его базовых классов;
- метод *не* должен иметь специальное имя.

Первые два критерия достаточно просты. Что касается третьего, который исключает методы, присутствующие в классе `Controller` и его базовых классах, то он означает, что исключаются методы вроде `ToString` и `GetHashCode`, а также методы, которые реализуют интерфейс `IController`. Это разумно, потому что мы не хотим демонстрировать внутренние элементы контроллеров. Последний критерий означает, что исключаются конструкторы, свойства и средства доступа к событиям – фактически из этого следует, что ни один член класса с флагом `IsSpecialName` из `System.Reflection.MethodBase` не будет использоваться для обработки действий.

## Примечание

*Методы, которые имеют общие параметры (такие как `MyMethod<T>()`), отвечают всем критериям, но при вызове такого метода для обработки запроса MVC Framework выбросит исключение.*

По умолчанию `ControllerActionInvoker` находит метод с таким же именем, как и у запрошенного действия. Так, например, если в системе маршрутизации свойство `action` содержит `Index`, то `ControllerActionInvoker` будет искать метод под названием `Index`, который соответствует критериям действия. Если он найдет такой метод, то вызовет его для обработки запроса. Такое поведение будет удовлетворять нашим нуждам в большинстве случаев, но, как и следовало ожидать, MVC Framework предоставляет некоторые возможности для тонкой настройки процесса.

## Переопределяем имя действия

Обычно имя метода действия определяет действие, которое он представляет. Метод действия `Index` обслуживает запросы к действию `Index`. Вы можете переопределить это поведение с помощью атрибута `ActionName`, который мы применили к контроллеру `Customer` в листинге 17-15.

### Листинг 17-15: Используем пользовательское имя действия

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

namespace ControllerExtensibility.Controllers
{
    public class CustomerController : Controller
    {
        public ViewResult Index()
```

```

    {
        return View("Result", new Result
    {
        ControllerName = "Customer",
        ActionName = "Index"
    });
}

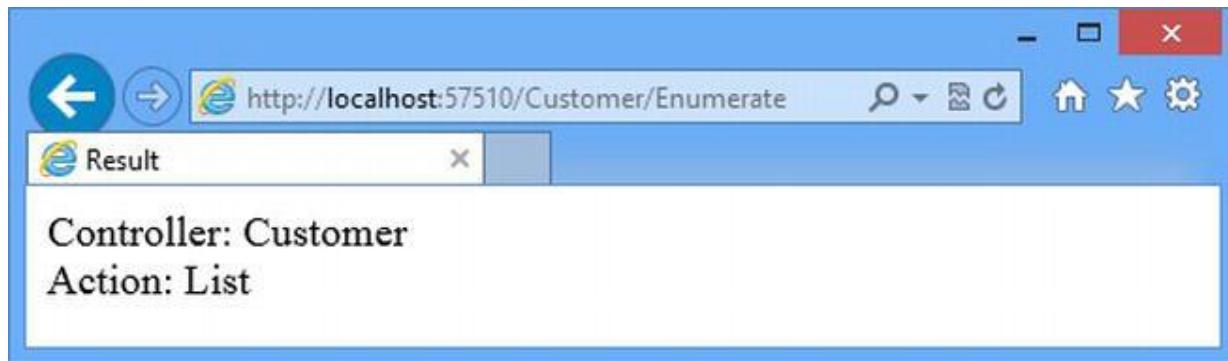
[ActionName("Enumerate")]
public ViewResult List()
{
    return View("Result", new Result
    {
        ControllerName = "Customer",
        ActionName = "List"
    });
}
}
}

```

В этом листинге мы применили атрибут к методу `List`, передав в параметр значение `Enumerate`. Когда механизм вызова действий получает запрос к действию `Enumerate`, для его обслуживания он теперь будет использовать метод `List`.

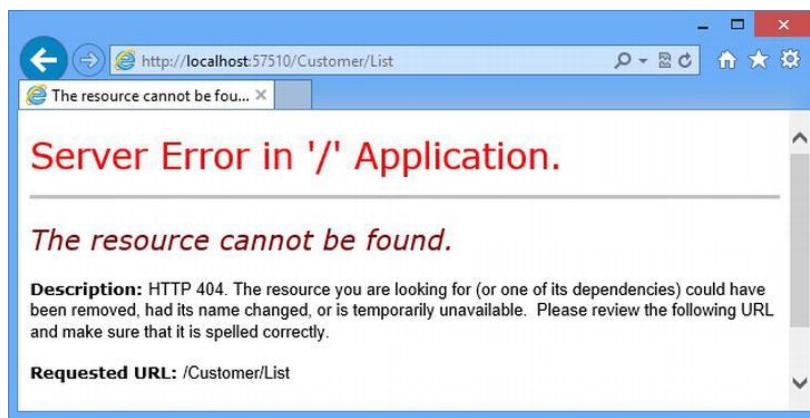
Вы можете увидеть эффект атрибута `ActionName`, запустив приложение и перейдя по ссылке `/Customer/Enumerate`. Как видите, результаты, показанные в браузере на рисунке 17-5, получены от метода `List`.

**Рисунок 17-5:** Эффект атрибута `ActionName`



Применение атрибута переопределяет имя действия. Это означает, что URL, которые ведут непосредственно к методу `List`, больше работать не будут, как показано на рисунке 17-6.

**Рисунок 17-6:** Используем имя метода в качестве действия после применения атрибута `ActionName`

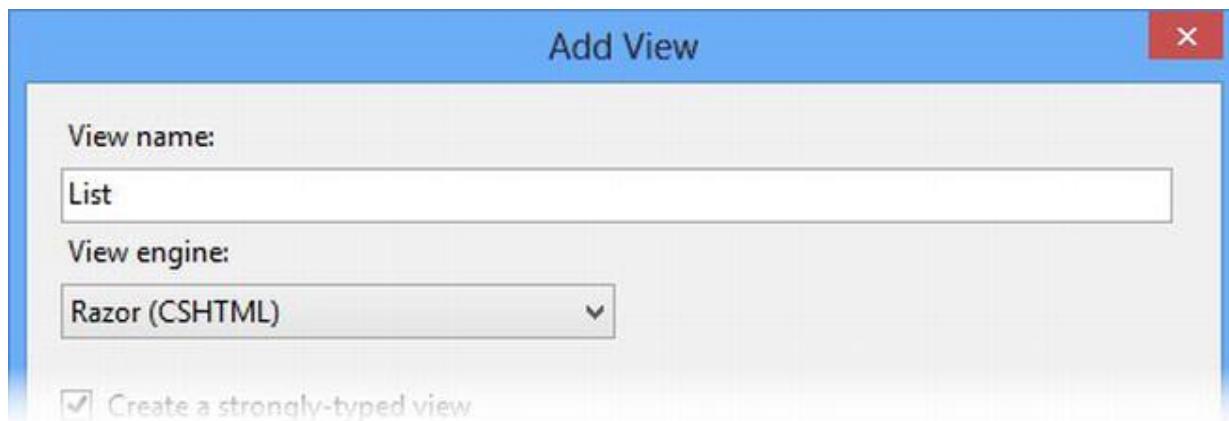


Существуют две основные причины переопределить таким образом имя метода.

- Чтобы использовать имена действий, которые не соответствуют правилам C# (Например, [ActionName ("User-Registration")]).
- Вам могут понадобиться два различных метода C#, которые принимают один и тот же набор параметров и должны обрабатывать одно и то же имя действия, но в ответ на различные типы запросов HTTP (например, один для [HttpGet], а другой для [HttpPost]). В таком случае вы можете дать методам различные имена C#, чтобы не возникло проблем с компилятором, но затем с помощью [ActionName] соотнести их с одним именем действия.

У этого атрибута есть один странный аспект: Visual Studio будет использовать имя оригинального метода в диалоговом окне Add View. Итак, если вы кликните правой кнопкой мыши по методу List и выберите пункт Add View, вы увидите диалоговое окно, показанное на рисунке 17-7.

**Рисунок 17-7:** Visual Studio не обнаруживает атрибут ActionName



Это проблема, потому что MVC Framework будет искать стандартные представления по имени действия, которым в нашем примере является List, что было определено в атрибуте. При создании представления по умолчанию для метода действия, который использует атрибут ActionName, вы должны убедиться, что его имя соответствует имени, указанному в атрибуте, а не методу C#.

## Используем селектор метода действия

Часто контроллер будет содержать несколько действий с одним и тем же именем. Так может получиться, если у вас есть несколько методов с различными параметрами, или если вы использовали атрибут ActionName, так что несколько методов представляют собой одно и то же действие.

В таких ситуациях MVC Framework требуется помочь при выборе соответствующего действия, которое должно обработать запрос. Механизм такой помощи называется селектором метода действия. Он позволяет определить виды запросов, которые может обрабатывать действие. Вы уже видели пример селектора метода действия, когда мы ограничили действие с помощью атрибута `HttpPost` в приложении SportsStore. У нас было два метода под названием `Checkout` в контроллере `Cart`, и с помощью `HttpPost` мы указали, который из них должен использоваться только для запросов HTTP POST, как показано в листинге 17-16.

**Листинг 17-16:** Используем атрибут `HttpPost`

```
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using SportsStore.WebUI.Models;
```

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace SportsStore.WebUI.Controllers
{
    public class CartController : Controller
    {
        // ...other instance members omitted for brevity...
        [HttpPost]
        public ViewResult Checkout(Cart cart, ShippingDetails shippingDetails)
        {
            if (cart.Lines.Count() == 0)
            {
                ModelState.AddModelError("", "Sorry, your cart is empty!");
            }
            if (ModelState.IsValid)
            {
                orderProcessor.ProcessOrder(cart, shippingDetails);
                cart.Clear();
                return View("Completed");
            }
            else
            {
                return View(shippingDetails);
            }
        }

        public ViewResult Checkout()
        {
            return View(new ShippingDetails());
        }
    }
}

```

Механизм вызова действий использует селекторы методов действий, чтобы избежать неоднозначности при выборе действий. В листинге 17-16 есть два кандидата для действия `Checkout`. Механизм вызова отдает предпочтение действиям с селекторами. В этом случае он смотрит на селектор `HttpPost`, чтобы решить, может ли запрос быть обработан данным методом. Если да, то механизм вызова выберет именно этот метод. Если нет, то будет использоваться его «конкурент».

Для различных видов HTTP-запросов существуют встроенные атрибуты, которые работают как селекторы: `HttpPost` для запросов `POST`, `HttpGet` для запросов `GET`, `HttpPut` для `PUT` и так далее. `NonAction` – это еще один встроенный атрибут, который указывает механизму вызова действий, что этот метод нельзя использовать, хотя он и является действительным методом действия. Мы применили атрибут `NonAction` в листинге 17-17, где мы также определили новый метод действия в контроллере `Customer`.

### Листинг 17-17: Используем селектор `NonAction`

```

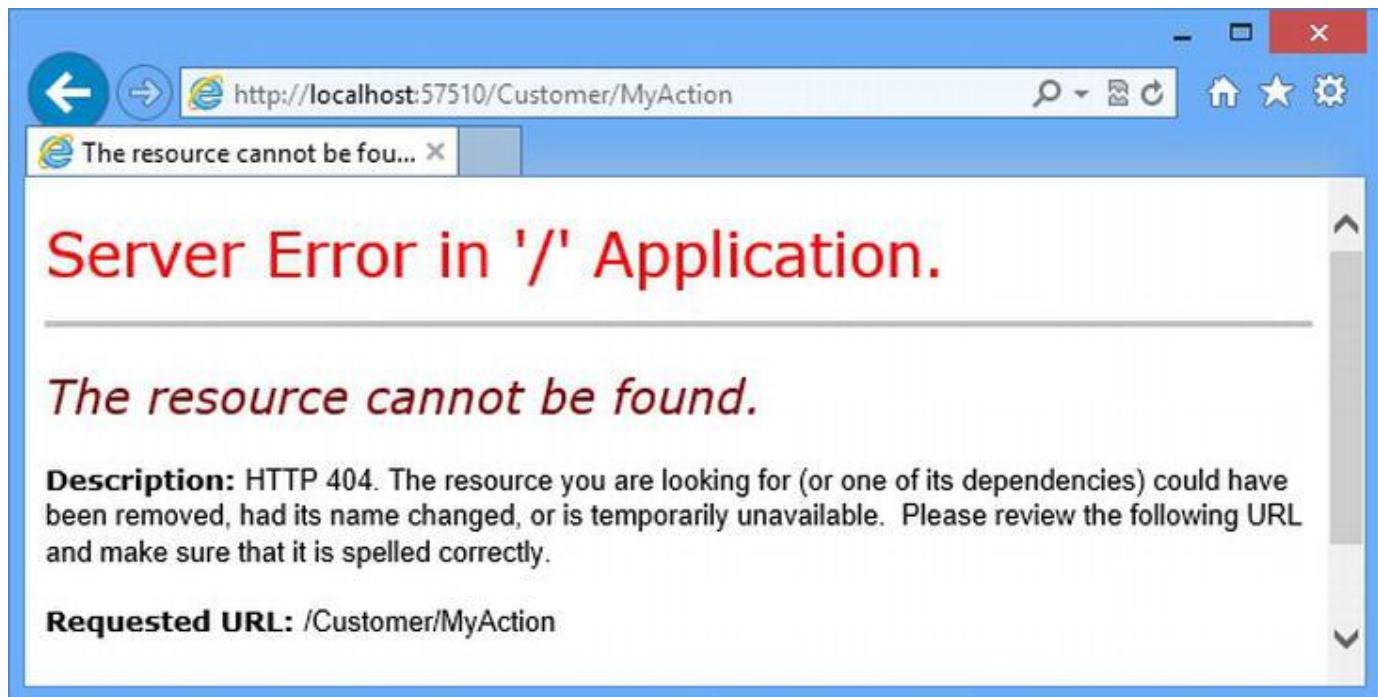
using ControllerExtensibility.Models;
using System.Web.Mvc;
namespace ControllerExtensibility.Controllers
{
    public class CustomerController : Controller
    {
        // ...other action methods omitted for brevity...
        [NonAction]
        public ActionResult MyAction()
        {

```

```
        return View();
    }
}
```

Метод `MyAction` в листинге не будет рассматриваться как метод действия, хотя он и отвечает всем критериям поиска механизма вызова. Таким образом можно гарантировать, что внутренние методы вашего приложения не будут использоваться как действия. Конечно, обычно такие методы просто должны иметь пометку `private`, которая не позволит вызывать их как действия, однако, если по каким-то причинам необходимо пометить такие методы как `public`, то придется использовать `[NonAction]`. Ссылки, которые ведут к методам `NonAction`, будут генерировать ошибки 404–Not Found, как показано на рисунке 17-8.

**Рисунок 17-8:** Эффект запроса URL, который ведет к методу `NonAction`



## Создаем пользовательский селектор метода действия

Селекторы методов действий наследуют от класса `ActionMethodSelectorAttribute`, который показан в листинге 17-18.

**Листинг 17-18:** Класс `ActionMethodSelectorAttribute`

```
using System.Reflection;
namespace System.Web.Mvc
{
    [AttributeUsage(AttributeTargets.Method, AllowMultiple = false, Inherited = true)]
    public abstract class ActionMethodSelectorAttribute : Attribute
    {
        public abstract bool IsValidForRequest(
            ControllerContext controllerContext,
            MethodInfo methodInfo);
    }
}
```

Класс `ActionMethodSelectorAttribute` является абстрактным и определяет один абстрактный метод: `IsValidForRequest`. Параметрами этого метода являются объект `ControllerContext`,

который передает информацию о запросе, и объект MethodInfo, с помощью которого можно получить информацию о методе, к которому применен селектор. IsValidForRequest возвращает true, если метод может обработать запрос, в противном случае - false. Мы создали простой пользовательский селектор метода действия под названием LocalAttribute в папке Infrastructure нашего проекта, как показано в листинге 17-19.

#### Листинг 17-19: Пользовательский селектор метода действия

```
using System.Reflection;
using System.Web.Mvc;
namespace ControllerExtensibility.Infrastructure
{
    public class LocalAttribute : ActionMethodSelectorAttribute
    {
        public override bool IsValidForRequest(
            ControllerContext controllerContext,
            MethodInfo methodInfo)
        {
            return controllerContext.HttpContext.Request.IsLocal;
        }
    }
}
```

Мы переопределили метод IsValidForRequest так, чтобы он возвращал true, когда запрос исходит из локальной машины. Чтобы продемонстрировать работу пользовательского селектора метода действия, мы создали в проекте контроллер Home, как показано в листинге 17-20.

#### Листинг 17-20: Контроллер Home

```
using System.Web.Mvc;
using ControllerExtensibility.Infrastructure;
using ControllerExtensibility.Models;

namespace ControllerExtensibility.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return View("Result", new Result
            {
                ControllerName = "Home",
                ActionName = "Index"
            });
        }

        [ActionName("Index")]
        public ActionResult LocalIndex()
        {
            return View("Result", new Result
            {
                ControllerName = "Home",
                ActionName = "LocalIndex"
            });
        }
    }
}
```

Мы использовали атрибут ActionName, чтобы создать ситуацию, в которой есть два метода действия Index. На данный момент механизм вызова действий не может выяснить, какой из них следует использовать для запроса ссылки /Home/Index, и при получении такого запроса будет генерироваться ошибка, показанную на рисунке 17-9.

**Рисунок 17-9:** Ошибка для неоднозначных имен методов действий



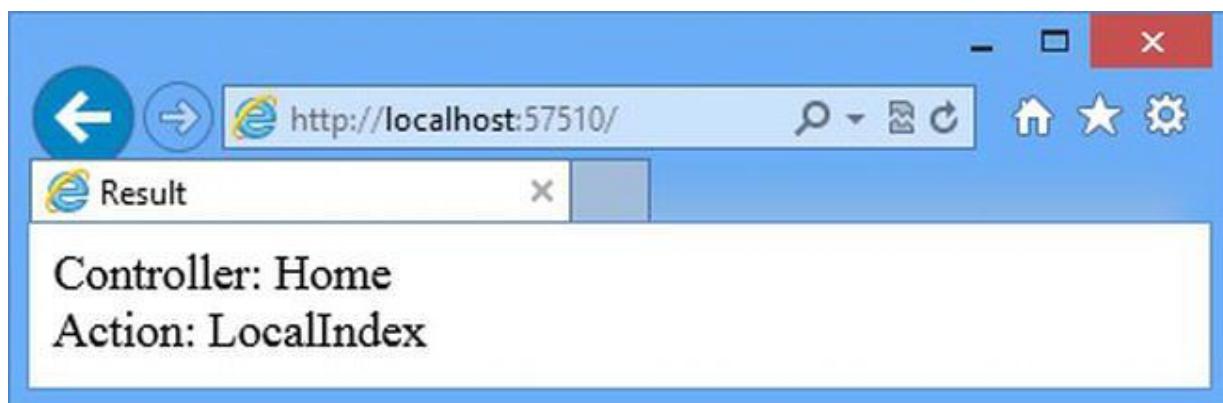
Чтобы разрешить эту ситуацию, мы можем применить атрибут селекции к одному из неоднозначных методов, как показано в листинге 17-21.

**Листинг 17-21:** Применяем атрибут селекции

```
[Local]
[ActionName("Index")]
public ActionResult LocalIndex()
{
    return View("Result", new Result
    {
        ControllerName = "Home",
        ActionName = "LocalIndex"
    });
}
```

Если вы перезапустите приложение и перейдите по корневой ссылке из браузера, работающего на локальной машине, вы увидите, что MVC Framework принимает во внимание атрибуты селекции, чтобы устранить неоднозначность методов в классе контроллера, как показано на рисунке 17-10.

**Рисунок 17-10:** Используем атрибут селекции для исключения неоднозначности методов действий



## Устранение неоднозначности методов действий

Теперь, когда вы видели изнанку базового класса селектора, вам будет легче понять, как механизм вызова действий выбирает метод действия. Механизм вызова сначала просматривает список методов контроллера, которые отвечают критериям и являются методами действий. Затем он проходит через следующий процесс:

Механизм вызова сортирует методы по имени. В списке остаются только методы с тем же именем, что и запрашиваемое действие, или с соответствующим атрибутом `ActionName`.

Механизм вызова отбрасывает методы с атрибутом селектора, который возвращает `false` для текущего запроса.

Если остается только один метод действия с селектором, то используется он. Если остается больше одного метода с селектором, то выбрасывается исключение, потому что механизм вызова не может устраниить неоднозначность между доступными методами.

Если методов с селектором не остается, то механизм вызова просматривает методы без селекторов. Если есть только один такой метод, то вызывается именно он. Если таких методов больше, то выбрасывается исключение, потому что механизм вызова не может выбрать один из них.

## Обработка неизвестных действий

Если механизм вызова действий не может найти подходящий метод действия, то его метод `InvokeAction` возвращает `false`. Когда это происходит, класс `Controller` вызывает метод `HandleUnknownAction`. По умолчанию этот метод возвращает ответ `404 – Not Found`. Такой ответ используется в большинстве приложений, но если вы хотите отображать что-то другое, то можете переопределить этот метод в каком-либо контроллере. В листинге 17-22 показано переопределение метода `HandleUnknownAction` в контроллере `Home`.

**Листинг 17-22:** Переопределяем метод `HandleUnknownAction`

```
using System.Web.Mvc;
using ControllerExtensibility.Infrastructure;
using ControllerExtensibility.Models;
namespace ControllerExtensibility.Controllers
{
    public class HomeController : Controller
    {
        // ...other action methods omitted for brevity...
        protected override void HandleUnknownAction(string actionPerformed)
        {
            Response.Write(string.Format("You requested the {0} action", actionPerformed));
        }
    }
}
```

Если вы запустите приложение и перейдите по ссылке, ведущей к несуществующему методу действия, то увидите ответ, показанный на рисунке 17-11.

**Рисунок 17-11:** Ответ для запросов к несуществующим методам действий



# Улучшение работы приложения при помощи специальных контроллеров

MVC Framework предоставляет два вида специализированных контроллеров, которые могут улучшить производительность вашего приложения MVC. Как и все средства оптимизации производительности, эти контроллеры представляют собой компромиссы: либо простоту использования, либо уменьшенную функциональностью. В последующих разделах мы рассмотрим оба вида контроллеров, опишем их преимущества и недостатки.

## Используем контроллеры без поддержки состояния сессии (sessionless controllers)

По умолчанию контроллеры поддерживают состояние сессии, в котором сохраняются данные между запросами, что облегчает разработку на MVC. Создание и поддержка состояния сессии – это комплексный процесс. Данные должны сохраняться и извлекаться, а сами сессии должны истекать в соответствующий момент. Данные сессии занимают память сервера или места в другие области памяти, и необходимость синхронизации данных на нескольких серверах затрудняет работу приложения на ферме серверов.

Чтобы упростить состояние сессии, ASP.NET обрабатывает только один запрос для данной сессии за один подход. Если клиент делает несколько перекрывающих друг друга во времени запросов, то они будут поставлены в очередь и обработаны последовательно. Преимущество этого подхода в том, что вам не придется беспокоиться о нескольких запросах, которые изменяют одни и те же данные. Его недостатком является то, что он снижает скорость обработки запросов.

Не всем контроллерам нужны данные состояния сессии. Для многих контроллеров можно убрать поддержку состояния сессии и улучшить производительность приложения. Для этого создаются контроллеры без поддержки состояния сессии. У них есть два отличия от обычных контроллеров: когда они обрабатывают запрос, MVC Framework не загружает и не сохраняет состояние сессии, а перекрывающие друг друга запросы обрабатываются одновременно.

## Управляем состоянием сессии в пользовательском IControllerFactory

Как уже упоминалось в начале этой главы, интерфейс `IControllerFactory` содержит метод `GetControllerSessionBehaviour`, который возвращает значение из перечисления `SessionStateBehaviour`. Это перечисление содержит четыре значения, которые определяют конфигурацию состояния сессии контроллера. Они описаны в таблице 17-3.

**Таблица 17-3:** Значения перечисления `SessionStateBehavior`

Значение	Описание
<code>Default</code>	Использует стандартное поведение ASP.NET, то есть определяет конфигурацию состояния сессии из <code>HttpContext</code> .
<code>Required</code>	Разрешает чтение и запись состояния сессии.
<code>ReadOnly</code>	Разрешает только чтение состояния сессии.
<code>Disabled</code>	Состояние сессии отключено.

Фабрика контроллеров, которая реализует интерфейс `IControllerFactory`, напрямую устанавливает поведение состояния сессии для контроллеров, возвращая значения `SessionStateBehavior` из метода `GetControllerSessionBehavior`. Параметрами этого метода являются объект `RequestContext` и `string`, содержащий имя контроллера. Вы можете вернуть любое из четырех

значений, приведенных в таблице, причем для разных контроллеров можно возвращать разные значения. Мы изменили реализацию метода `GetControllerSessionBehavior` в классе `CustomControllerFactory`, который создали ранее в данной главе, как показано в листинге 17-23.

#### Листинг 17-23: Определяем поведение состояния сессии для контроллера

```
public SessionStateBehavior GetControllerSessionBehavior(
    RequestContext requestContext,
    string controllerName)
{
    switch (controllerName)
    {
        case "Home":
            return SessionStateBehavior.ReadOnly;
        case "Product":
            return SessionStateBehavior.Required;
        default:
            return SessionStateBehavior.Default;
    }
}
```

#### Управляем состоянием сессии с помощью DefaultControllerFactory

Используя встроенную фабрику контроллеров, вы можете управлять состоянием сеанса, применяя атрибут `SessionState` к отдельным классам контроллеров, как показано в листинге 17-24 на примере нового контроллера `FastController`.

#### Листинг 17-24: Используем атрибут `SessionState`

```
using System.Web.Mvc;
using System.Web.SessionState;
using ControllerExtensibility.Models;
namespace ControllerExtensibility.Controllers
{
    [SessionState(SessionStateBehavior.Disabled)]
    public class FastController : Controller
    {
        public ActionResult Index()
        {
            return View("Result", new Result
            {
                ControllerName = "Fast ",
                ActionName = "Index"
            });
        }
    }
}
```

Атрибут `SessionState` применяется к классу контроллера и влияет на все его действия. Единственный параметр атрибута – это значение из перечисления `SessionStateBehavior` (см. таблицу 17-3). В этом примере мы отключили состояние сессии, что означает, что если мы попытаемся установить в контроллере значение сессии:

```
Session["Message"] = "Hello";
```

или прочитать информацию из состояния сессии в представлении:

```
Message: @Session["Message"]
```

MVC Framework выбросит исключение при вызове действия или визуализации представления.

### *Подсказка*

*Когда состояние сеанса отключено, свойство `HttpContext.Session` возвращает значение `null`.*

---

Если вы указали поведение `ReadOnly`, то сможете прочитать значения, установленные другими контроллерами, но все равно получите исключение среди выполнения, если попытаетесь их установить или изменить. Подробную информацию о сессии можно получить через объект `HttpContext.Session`, но попытка изменить значения приведет к ошибке.

### *Подсказка*

*Если вы просто хотите передать данные из контроллера в представление, рекомендуется использовать объект `ViewBag`, на который атрибут `SessionState` не влияет.*

---

## **Используем асинхронные контроллеры**

Базовая платформа ASP.NET поддерживает пул потоков .NET, которые используются для обработки клиентских запросов. Этот пул называется пулем рабочих потоков (*worker thread pool*), а потоки, соответственно, рабочими (*worker threads*). При получении запроса рабочий поток вызывается из пула и обрабатывает запрос.

После обработки запроса рабочий поток возвращается в пул и может обрабатывать новые запросы по мере их поступления. Пулы потоков имеют два ключевых преимущества для приложений ASP.NET:

- Повторное использование рабочих потоков избавляет от необходимости создавать новый поток для каждого запроса.
- Наличие фиксированного числа рабочих потоков позволяет избежать ситуации, когда вы обрабатываете больше одновременных запросов, чем может сервер.

Пул рабочих потоков работает лучше всего, когда запросы обрабатываются в течение короткого промежутка времени. Так бывает в большинстве приложений MVC. Тем не менее, если у вас есть действия, которые зависят от других серверов и их обработка занимает более длительный период времени, то вы можете попасть в ситуацию, где все ваши рабочие потоки связаны ожиданием ответов от других систем.

### *Примечание*

*В этом разделе мы предполагаем, что вы знакомы с Task Parallel Library (TPL). Если вы хотите узнать больше о TPL, прочтите книгу Адама на эту тему, *Pro .NET Parallel Programming in C#* издательства Apress.*

---

Ваш сервер способен делать больше - в конце концов, вы только ждете, и это не потребляет много ресурсов. Но, так как все ваши рабочие потоки связаны, входящие запросы ставятся в очередь. Вы попадете в странную ситуацию: приложение зависает, но сервер практически бездействует.

### **Внимание!**

*В этот момент некоторые читатели могут подумать, что можно написать пул рабочих потоков, адаптированный к их приложению. Не стоит этого делать. Написать код параллельного действия легко. Написать код параллельного действия, который будет работать, трудно. Если вы новичок в параллельном программировании, то вам не хватит навыка. Наш совет: используйте стандартный пул. Если у вас большой опыт в параллельном программировании, то вы уже понимаете, что выигрыш будет незначительным по сравнению с усилиями, которые будут затрачены на кодирование и тестирование нового пула потоков.*

Решение этой проблемы представляют асинхронные контроллеры. Они увеличивают общую производительность приложения, хотя и не приносят никакой пользы для выполнения асинхронных операций.

### **Примечание**

Асинхронные контроллеры полезны только для действий, которые ориентированы на ввод/вывод или передачу данных в сети и не зависят от быстродействия процессора. Асинхронные контроллеры помогают решить проблему с несоответствием между моделью пула и типом обрабатываемого запроса. Пул должен гарантировать то, что каждый запрос получит необходимые ресурсы сервера, но в итоге вы получаете набор рабочих потоков, которые ничего не делают. Если вы используете дополнительные фоновые потоки для действий с высокой нагрузкой на процессор, то сможете распределить ресурсы сервера между большим числом одновременных запросов.

### **Создаем пример**

Чтобы начать изучение асинхронных контроллеров, мы продемонстрируем вам проблему, которую они могут решить. В листинге 17-25 показан обычный синхронный контроллер под названием `RemoteData`, который мы добавили в проект.

#### **Листинг 17-25:** Проблемный синхронный контроллер

```
using System.Web.Mvc;
using ControllerExtensibility.Models;
namespace ControllerExtensibility.Controllers
{
    public class RemoteDataController : Controller
    {
        public ActionResult Data()
        {
            RemoteService service = new RemoteService();
            string data = service.GetRemoteData();
            return View((object)data);
        }
    }
}
```

Этот контроллер содержит метод действия, `Data`, который создает экземпляр класса модели `RemoteService` и вызывает в нем метод `GetRemoteData`. Этот метод отнимает много времени и снижает активность процессора. Класс `RemoteService` показан в листинге 17-26.

#### **Листинг 17-26:** Сущность модели с помощью трудоемким методом

```
using System.Threading;
namespace ControllerExtensibility.Models
```

```

{
    public class RemoteService
    {
        public string GetRemoteData()
        {
            Thread.Sleep(2000);
            return "Hello from the other side of the world";
        }
    }
}

```

Okay, здесь мы просто подделали метод GetRemoteData. В реальном мире этот метод мог бы извлекать сложные данные через медленное сетевое подключение, но для простоты мы использовали метод Thread.Sleep для моделирования двух-секундной задержки. Мы также создали простое представление под названием Data.cshtml, которое показано в листинге 17-27.

**Листинг 17-27:** Представление Data

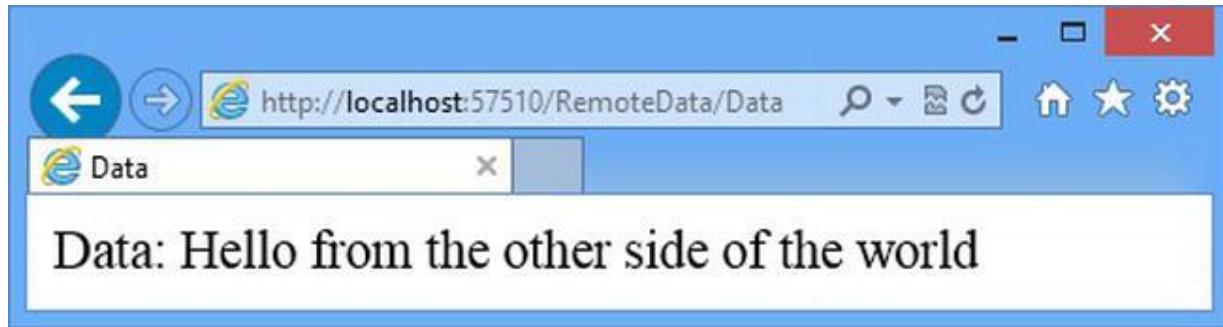
```

@model string
 @{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Data</title>
</head>
<body>
    <div>
        Data: @Model
    </div>
</body>
</html>

```

Если вы запустите приложение и перейдите по ссылке RemoteData/Data, будет вызван метод действия, создан объект RemoteService и отправлен вызов к методу GetRemoteData. Через две секунды (которые имитируют выполнение реальной операции) метод GetRemoteData возвращает данные, они передаются представлению и визуализируются, как показано на рисунке 17-12.

**Рисунок 17-12:** Переход по ссылке /RemoteData/Data



Проблема здесь в том, что рабочий поток, который обрабатывал наш запрос, бездействовал в течение двух секунд – он не делал ничего полезного, но и не был доступен для обработки других запросов.

### Создаем асинхронный контроллер

Продемонстрировав вам проблему, мы можем теперь перейти к ее решению и создать асинхронный контроллер. Для этого есть только два способа. Первый – реализовать интерфейс

`System.Web.Mvc.Async.IAsyncController`, который является асинхронным эквивалентом `IController`. Мы не собираемся разбирать данный подход, потому что для этого потребуется объяснять много функций параллельного программирования .NET.

#### *Подсказка*

*Не все действия в асинхронных контроллерах должны быть асинхронными. Вы можете включать в них и синхронные методы, и они будут работать нормально.*

Мы хотим, чтобы в центре нашего внимания оставался MVC Framework, и поэтому продемонстрируем второй подход: наследовать класс контроллера от `System.Web.Mvc.AsyncController`, который реализует `IAsyncController`. В предыдущей версии .NET Framework создание асинхронных контроллеров было сложным процессом и требовало разделения действия на два метода. Новые ключевые слова `await` и `async`, о которых мы рассказывали в главе 4, намного упростили этот процесс: вы создаете новый объект `Task` и ждете его ответа, как показано в листинге 17-28.

#### *Подсказка*

*Старая техника создания асинхронных методов действий все еще поддерживается, хотя подход, который мы описываем здесь, гораздо проще и надежнее. Единственным напоминанием о старом подходе является то, что вы не можете использовать имена методов действий, которые заканчиваются на `Async` (например `IndexAsync`) или `Completed` (например `IndexCompleted`).*

**Листинг 17-28:** Создаем асинхронный контроллер без изменения вызываемых им методов

```
using System.Web.Mvc;
using ControllerExtensibility.Models;
using System.Threading.Tasks;

namespace ControllerExtensibility.Controllers
{
    public class RemoteDataController : AsyncController
    {
        public async Task<ActionResult> Data()
        {
            string data = await Task<string>.Factory.StartNew(() =>
            {
                return new RemoteService().GetRemoteData();
            });
            return View((object)data);
        }
    }
}
```

Мы выделили изменения, которые делают контроллер `RemoteData` асинхронным. Кроме изменения базового класса `AsyncController`, мы провели рефакторинг метода действия так, чтобы он возвращал `Task<ActionResult>`, применили ключевые слова `async` и `await` и создали `Task<string>`, который будет вызывать метод `GetRemoteData`.

## Используем асинхронные методы в контроллере

С помощью асинхронных контроллеров можно использовать асинхронные методы где угодно в приложении. Чтобы это продемонстрировать, мы добавили асинхронный метод в класс `RemoteService`, как показано в листинге 17-29.

### Листинг 17-29: Добавляем асинхронный метод в класс `RemoteService`

```
using System.Threading;
using System.Threading.Tasks;

namespace ControllerExtensibility.Models
{
    public class RemoteService
    {
        public string GetRemoteData()
        {
            Thread.Sleep(2000);
            return "Hello from the other side of the world";
        }

        public async Task<string> GetRemoteDataAsync()
        {
            return await Task<string>.Factory.StartNew(() =>
            {
                Thread.Sleep(2000);
                return "Hello from the other side of the world";
            });
        }
    }
}
```

Результатом метода `GetRemoteDataAsync` является `Task<string>`, который производит такое же сообщение, как и синхронный метод по завершении. В листинге 17-30 вы можете увидеть, как мы использовали этот асинхронный метод действия в новом методе, который добавили в контроллер `RemoteData`.

### Листинг 17-30: Используем асинхронный метод в контроллере `RemoteData`

```
using System.Web.Mvc;
using ControllerExtensibility.Models;
using System.Threading.Tasks;

namespace ControllerExtensibility.Controllers
{
    public class RemoteDataController : AsyncController
    {
        public async Task<ActionResult> Data()
        {
            string data = await Task<string>.Factory.StartNew(() =>
            {
                return new RemoteService().GetRemoteData();
            });
            return View((object)data);
        }

        public async Task<ActionResult> ConsumeAsyncMethod()
        {
            string data = await new RemoteService() .GetRemoteDataAsync();
            return View("Data", (object)data);
        }
    }
}
```

Как видите, оба метода действий следуют одному и тому же базовому шаблону и отличаются тем, где создается объект `task`. В результате вызова любого из этих методов действий рабочий поток не будет связан, пока мы ждем завершения вызова `GetRemoteData`. Это означает, что поток сможет обрабатывать другие запросы, что может значительно улучшить производительность приложения.

## Резюме

В этой главе мы рассмотрели, как MVC Framework создает контроллеры и вызывает методы. Мы исследовали и настроили встроенные реализации ключевых интерфейсов и создали их пользовательские версии, чтобы продемонстрировать, как они работают. Вы узнали, как можно использовать селекторы методов действий для устранения неоднозначности, и познакомились с некоторыми специализированными видами контроллеров, которые могут расширить возможности обработки запросов в приложении.

Основной темой этой главы является расширяемость. Почти каждый компонент MVC Framework можно изменить или полностью заменить. Для большинства проектов поведение по умолчанию является вполне достаточным. Но практические знания о том, как работает MVC Framework, помогут вам принимать более обоснованные решения относительно дизайна и кода приложения.

# Представления

Как вы узнали из главы 15, методы действий могут возвращать объекты результатов действий `ActionResult`; наиболее часто используемым результатом действия является `ViewResult`, который запускает визуализацию представления и отправляет его клиенту.

Вы уже примерно знаете, для чего нужны представления, ведь мы часто использовали их в примерах приложений. В данной главе мы углубим и уточним эти знания. Для начала мы рассмотрим, как MVC Framework обрабатывает объекты `ViewResult` с помощью *движков представлений* (*view engine*), а также научимся создавать пользовательские движки представлений. Далее будут описаны эффективные способы работы со встроенным движком Razor View Engine. В продолжение мы продемонстрируем, как создавать и использовать частичные представления, дочерние действия и секции Razor. Все вышеперечисленные темы очень важны для эффективной работы с MVC.

## Создание пользовательского движка представления

Эту главу мы начнем с создания пользовательского движка представлений, хотя в реальных проектах он вам понадобится очень редко. MVC Framework включает в себя два встроенных движка представлений, полнофункциональных и тщательно протестированных:

- Движок Razor, который мы уже использовали в этой книге, появился в третьей версии MVC. У него простой и удобный синтаксис, который мы рассмотрели в главе 5.
- Движок ASPX, также известный как движок представлений Web Forms, использует синтаксис тегов Web Forms `<% ...%>`. Этот движок используется для поддержки совместимости в старых приложениях MVC.

Весь смысл создания пользовательского движка представлений состоит в том, чтобы с его помощью продемонстрировать работу конвейера обработки запросов и дополнить ваши знания об устройстве MVC Framework. Вы сможете оценить, какой гибкостью обладает процесс преобразования объектов `ViewResult` в ответ клиенту. Движки представлений реализуют интерфейс `IViewEngine`, который показан в листинге 18-1.

**Листинг 18-1:** Интерфейс `IViewEngine`

```
namespace System.Web.Mvc
{
    public interface IViewEngine
    {
        ViewEngineResult FindPartialView(ControllerContext controllerContext,
            string partialViewName,
            bool useCache);

        ViewEngineResult FindView(ControllerContext controllerContext,
            string viewName,
            string masterName,
            bool useCache);

        void ReleaseView(ControllerContext controllerContext, IView view);
    }
}
```

Задача движка представлений заключается в преобразовании запросов к представлениям в объекты `ViewEngineResult`. Первые два метода в интерфейсе, `FindView` и `FindPartialView`, принимают

параметры, в которых указывается запрос и обработавший его контроллер (объект ControllerContext), имя представления и его макет, а также сообщение о том, может ли движок представлений использовать предыдущий результат из кэша. Эти методы вызываются во время обработки ViewResult. Последний метод, ReleaseView, вызывается, когда представление больше не требуется.

#### Примечание

*Поддержка движков представлений в MVC Framework реализуется классом ControllerActionInvoker, который является встроенной реализацией интерфейса IActionInvoker, описанного в главе 15. Вы не сможете автоматически использовать движки представлений, если вы реализовали пользовательский механизм вызова действий или фабрику контроллеров непосредственно из интерфейсов IActionInvoker или IControllerFactory.*

---

Класс ViewEngineResult позволяет движку представлений отвечать на запрос представления. Он показан в листинге 18-2.

**Листинг 18-2:** Класс ViewEngineResult

```
using System.Collections.Generic;

namespace System.Web.Mvc
{
    public class ViewEngineResult
    {
        public ViewEngineResult(IEnumerable<string> searchedLocations)
        {
            if (searchedLocations == null)
            {
                throw new ArgumentNullException("searchedLocations");
            }
            SearchedLocations = searchedLocations;
        }

        public ViewEngineResult(IView view, IViewEngine viewEngine)
        {
            if (view == null)
            {
                throw new ArgumentNullException("view");
            }
            if (viewEngine == null)
            {
                throw new ArgumentNullException("viewEngine");
            }
            View = view;
            ViewEngine = viewEngine;
        }

        public IEnumerable<string> SearchedLocations { get; private set; }
        public IView View { get; private set; }
        public IViewEngine ViewEngine { get; private set; }
    }
}
```

Для создания результата необходимо выбрать один из двух конструкторов. Если ваш движок возвращает представления в ответ на запросы, то ViewEngineResult создается помошью следующего конструктора:

```
public ViewEngineResult(IView view, IViewEngine viewEngine)
```

Его параметрами являются реализация интерфейса `IView` и движок представления (так что позже будет вызван метод `ReleaseView`). Если ваш движок не возвращает представления на запросы, то используйте следующий конструктор:

```
public ViewEngineResult(IEnumerable<string> searchedLocations)
```

Здесь параметром является перечисление мест для поиска представления. Если представление не может быть найдено, пользователю будет показано соответствующее сообщение, которое мы рассмотрим позже.

#### *Примечание*

*Вам не кажется, что класс `ViewEngineResult` какой-то громоздкий? Выводить результаты с помощью разных версий конструктора класса – это нехарактерный подход для MVC. Мы не знаем, почему разработчики остановились на этом решении, но, к сожалению, вынуждены с ним мириться. Были надежды, что данный класс будет изменен в MVC 4, но этого не произошло.*

---

Последним структурным элементом движка представления является интерфейс `IView`, который показан в листинге 18-3.

#### **Листинг 18-3:** Интерфейс `IView`

```
using System.IO;

namespace System.Web.Mvc
{
    public interface IView
    {
        void Render(ViewContext viewContext, TextWriter writer);
    }
}
```

Мы передаем реализацию `IView` в конструктор объекта `viewEngineResult`, который будет позже возвращен из методов движка представления. MVC Framework вызывает метод `Render`. Параметр `ViewContext` предоставляет информацию о запросе клиента и выводе метода действия. Параметр `TextWriter` записывает ответ клиенту.

Как мы уже говорили, самый простой способ изучить работу всех этих элементов - `IViewEngine`, `IView` и `ViewEngineResult` - это создать пользовательский движок представлений. Мы напишем простой движок, который возвращает только один вид представления. Оно будет визуализировать результат, который содержит информацию о запросе и данные представления, сгенерированные методом действия. Таким образом мы сможем продемонстрировать, как работают движки представлений, не погружаясь в анализ шаблонов представлений.

### **Создаем пример проекта**

В качестве примера для этой главы мы создали проект под названием `Views`, используя шаблон `Empty`. В нем был создан контроллер `Home`, который вы можете увидеть в листинге 18-4.

#### Листинг 18-4: Контроллер Home в проекте Views

```
using System;
using System.Web.Mvc;

namespace Views.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewData["Message"] = "Hello, World";
            ViewData["Time"] = DateTime.Now.ToShortTimeString();
            return View("DebugData");
        }

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

Для этого проекта мы не создали ни одного представления, потому что собираемся реализовывать пользовательский движок, а не использовать Razor.

#### Создаем пользовательский IView

Начнем с создания пользовательской реализации `IView`. Мы добавили в пример проекта папку `Infrastructure` и создали в ней класс под названием `DebugDataView`, который показан в листинге 18-5.

#### Листинг 18-5: Пользовательская реализация `IView`

```
using System.IO;
using System.Web.Mvc;

namespace Views.Infrastructure
{
    public class DebugDataView : IView
    {
        public void Render(ViewContext viewContext, TextWriter writer)
        {
            Write(writer, "---Routing Data---");
            foreach (string key in viewContext.RouteData.Values.Keys)
            {
                Write(writer, "Key: {0}, Value: {1}", key, viewContext.RouteData.Values[key]);
            }
            Write(writer, "---View Data---");
            foreach (string key in viewContext.ViewData.Keys)
            {
                Write(writer, "Key: {0}, Value: {1}", key, viewContext.ViewData[key]);
            }
        }

        private void Write(TextWriter writer, string template, params object[] values)
        {
            writer.Write(string.Format(template, values) + "<p/>");
        }
    }
}
```

Как видите, в методе `Render` этого представления используются два параметра: мы берем значения из `ViewContext` и записываем ответ клиенту с помощью `TextWriter`. Немного позже вы увидите, как функционирует этот класс.

## Создаем реализацию `IViewEngine`

Следует помнить, что главная задача движка представления - создавать объект `ViewEngineResult`, который содержит либо `IView`, либо список мест для поиска подходящего представления. Теперь, когда у нас есть реализация `IView`, мы можем создать движок представления. В папке `Infrastructure` мы создали новый класс под названием `DebugDataViewEngine.cs`, содержание которого приведено в листинге 18-6.

**Листинг 18-6:** Пользовательская реализация `IViewEngine`

```
using System.Web.Mvc;

namespace Views.Infrastructure
{
    public class DebugDataViewEngine : IViewEngine
    {
        public ViewEngineResult FindView(ControllerContext controllerContext,
            string viewName,
            string masterName,
            bool useCache)
        {
            if (viewName == "DebugData")
            {
                return new ViewEngineResult(new DebugDataView(), this);
            }
            else
            {
                return new ViewEngineResult(new string[] {"No view (Debug Data View Engine)"});
            }
        }

        public ViewEngineResult FindPartialView(ControllerContext controllerContext,
            string partialViewName,
            bool useCache)
        {
            return new ViewEngineResult(new string[] {"No view (Debug Data View Engine)"});
        }

        public void ReleaseView(ControllerContext controllerContext, IView view)
        {
            // do nothing
        }
    }
}
```

Мы реализуем поддержку только одного представления, которое называется `DebugData`. При получении запроса к нему мы будем возвращать экземпляр пользовательской реализации `IView`, как показано далее:

```
return new ViewEngineResult(new DebugDataView(), this);
```

Если бы мы реализовывали более серьезный движок, то использовали бы возможность поиска шаблонов, учитывали макет и предоставляли параметры кэширования. Но в нашем простом примере требуется только создавать новый экземпляр класса `DebugDataView`. Если мы получим запрос к другому представлению (не `DebugData`), то вернем `ViewEngineResult`, как показано далее:

```
return new ViewEngineResult(new string[] { "No view (Debug Data View Engine)" });
```

Интерфейс `IViewEngine` предполагает, что движок знает, где искать представления. Это разумное предположение, так как представления обычно представляют собой шаблоны, которые хранятся в проекте как отдельные файлы. В данном случае нам не нужно ничего искать, и мы просто возвращаем фиктивное месторасположение, указывающее на то, что мы не можем найти запрошенное представление.

Наш пользовательский движок не поддерживает частичные представления, поэтому метод `FindPartialView` тоже будет возвращать результат, указывающий, что мы не можем найти нужное представление. Далее в этой главе мы вернемся к частичным представлениям и рассмотрим, как они обрабатываются в движке Razor. Метод `ReleaseView` пока еще ничего не делает, потому что в нашей реализации `IView` еще нет ресурсов, с которыми он мог бы работать.

## Регистрируем пользовательский движок представлений

Движок представлений регистрируется в методе `Application_Start` файла `Global.asax`, как показано в листинге 18-7.

**Листинг 18-7:** Регистрируем пользовательский движок представлений в `Global.asax`

```
using System.Web.Http;
using System.Web.Mvc;
using System.Web.Routing;
using Views.Infrastructure;

namespace Views
{
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();

            ViewEngines.Engines.Add(new DebugDataViewEngine());

            WebApiConfig.Register(GlobalConfiguration.Configuration);
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
        }
    }
}
```

MVC Framework поддерживает возможность установки нескольких движков в одном приложении – их перечень содержится в статической коллекции `ViewEngine.Engines`. При обработке `ViewResult` механизм вызова действий получает набор установленных движков и вызывает их методы `FindView` по очереди.

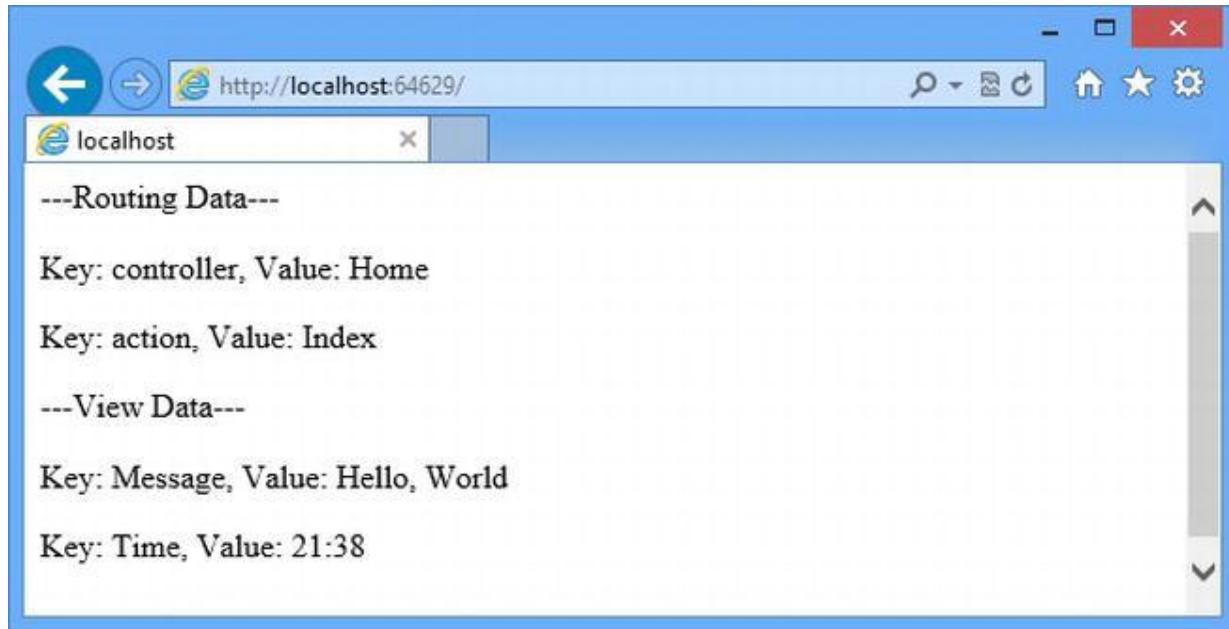
Когда механизм вызова действий получает объект `ViewEngineResult`, содержащий `IView`, он прекращает вызывать методы `FindView`. Это значит, что если два и более движка могут обслужить запрос к одному представлению, то решающее значение имеет последовательность, в которой они добавлены в коллекцию `ViewEngines.Engines`. Если вы хотите назначить вашему движку более высокий приоритет, вставьте его в начале коллекции, как показано здесь:

```
ViewEngines.Engines.Insert(0, new DebugDataViewEngine());
```

## Тестируем движок представлений

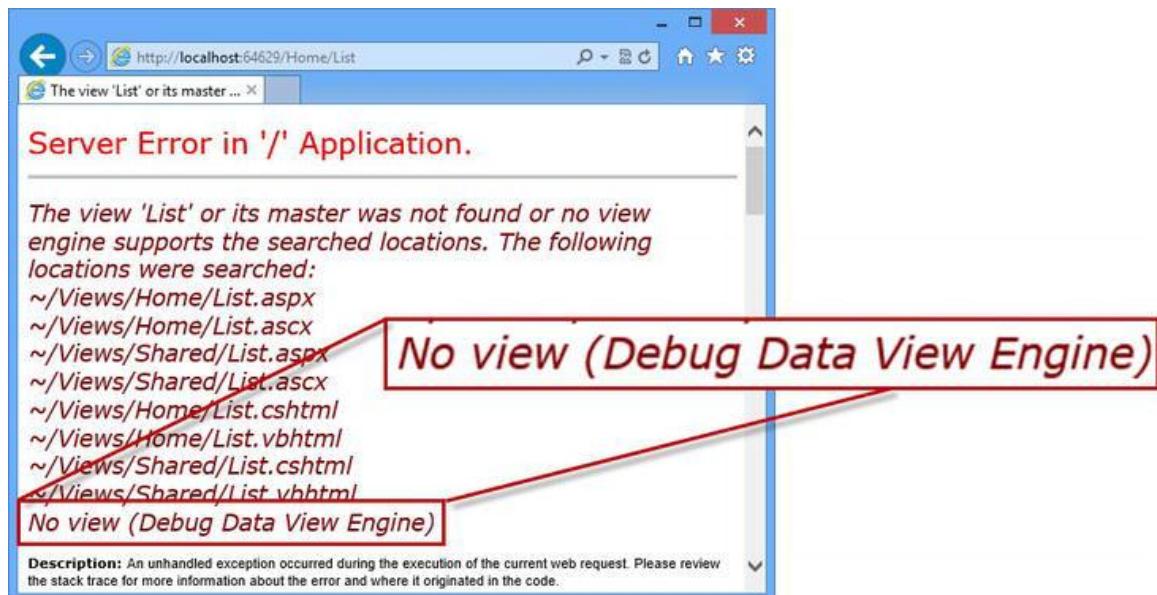
Сейчас мы уже в состоянии протестировать наш движок представлений. Если запустить приложение, браузер автоматически перейдет по корневому URL проекта, который будет соотнесен с действием Index контроллера Home. Используя метод View, метод действия возвращает ViewResult, который содержит представление DebugData. Результат показан на рисунке 18-1.

Рисунок 18-1: Использование пользовательского движка представлений



Это результат вызова метода FindView для представления, которое мы в состоянии обработать. Если мы перейдем по ссылке /Home/List, MVC Framework вызовет метод действия List, который вызовет метод View для запроса своего представления по умолчанию, которое мы не поддерживаем. Результат показан на рисунке 18-2.

Рисунок 18-2: Запрос неподдерживаемого представления



Как видите, в нашем сообщении отображается список адресов для поиска представления. Обратите внимание, что представления Razor и ASPX также появляются в списке, так как эти движки по-прежнему используются. Если мы хотим использовать только наш пользовательский движок, то перед его регистрацией в файле Global.asax нужно вызвать метод Clear, как показано в листинге 18-8.

**Листинг 18-8:** Удаляем другие движки представлений

```
using System.Web.Http;
using System.Web.Mvc;
using System.Web.Routing;
using Views.Infrastructure;

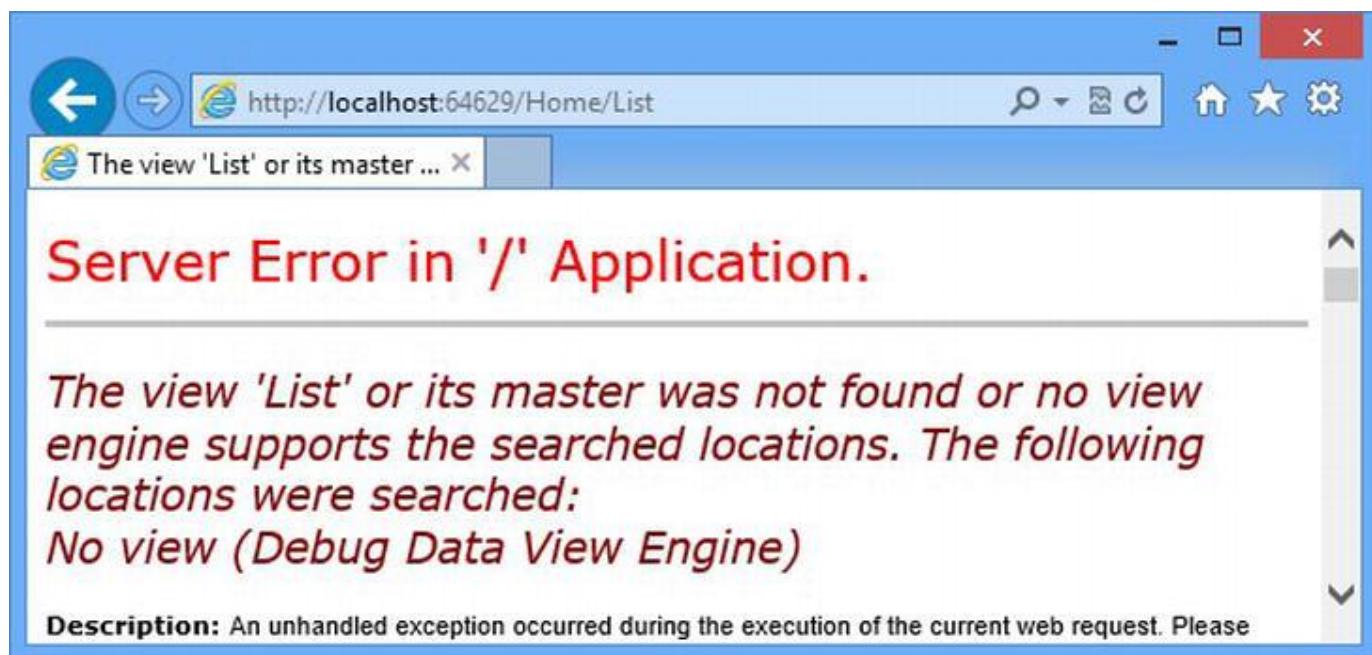
namespace Views
{
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();

ViewEngines.Engines.Clear();
            ViewEngines.Engines.Add(new DebugDataViewEngine());

            WebApiConfig.Register(GlobalConfiguration.Configuration);
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
        }
    }
}
```

Если мы перезапустим приложение и перейдем по ссылке /Home/List, будет использоваться только наш пользовательский движок, как показано на рисунке 18-3.

**Рисунок 18-3:** Использование только пользовательского движка



# Работа с движком представления Razor

В предыдущем разделе мы смогли создать пользовательский движок представлений, реализовав всего два интерфейса. Следует признать, что он получился довольно незамысловатым и генерирует упрощенные представления, но прекрасно иллюстрирует концепцию расширяемости MVC на примере конвейера обработки запросов.

Всю сложность движку представлений добавляет система шаблонов представлений, которые включают в себя фрагменты кода и поддержку макетов и служат для оптимизации производительности. Мы не включали такие компоненты в наш пользовательский движок – они там и не нужны, так как все необходимое включено в движок Razor. В Razor доступна функциональность, которая требуется практически во всех приложениях MVC, и создавать пользовательский движок понадобится только в очень редких случаях.

Напомним, что Razor появился в MVC 3 и заменил предыдущий движок представлений (известный как ASPX или движок Web Forms). Вы можете использовать ASPX, но мы рекомендуем остановиться на Razor, так как Microsoft в данный момент работает именно с ним. Основы синтаксиса Razor были рассмотрены в главе 5. В этой главе мы продемонстрируем вам другие функции для создания и отображения представлений Razor, а также разберем его настройки.

## Создаем пример проекта

Для этой части главы мы создали проект WorkingWithRazor с помощью шаблона Basic. В нем был создан контроллер Home, который показан в листинге 18-9.

### Листинг 18-9: Контроллер Home в проекте WorkingWithRazor

```
using System.Web.Mvc;

namespace WorkingWithRazor.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            string[] names = {"Apple", "Orange", "Pear"};
            return View(names);
        }
    }
}
```

Для этого контроллера мы создали представление Index.cshtml, которое поместили в папку /Views/Home. Содержимое файла представления показано в листинге 18-10.

### Листинг 18-10: Содержимое файла View.cshtml

```
@model string []

@{
    ViewBag.Title = "Index";
}

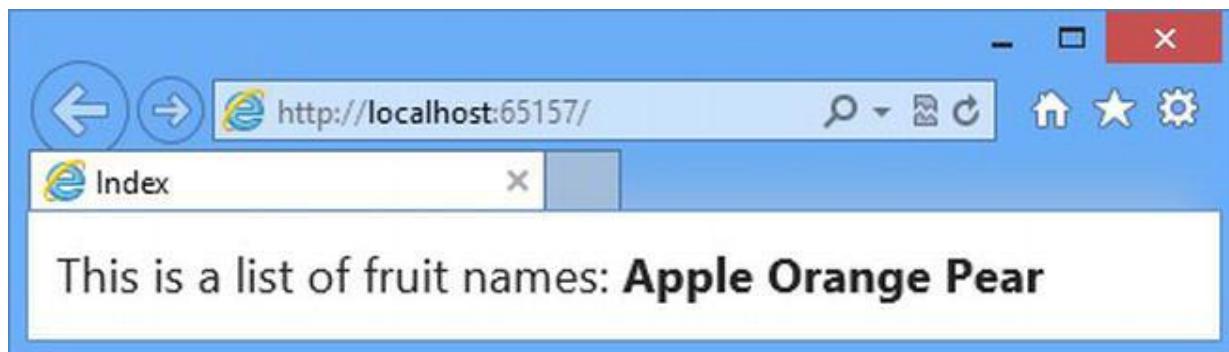
This is a list of fruit names:
@foreach (string name in Model)
{
    <span><b>@name</b></span>
}
```

## Основы визуализации представлений в Razor

Движок Razor компилирует представления в приложении для повышения производительности. Представления преобразуются в классы C#, а затем компилируются - именно поэтому в них можно легко включать фрагменты кода C#. Полезно просматривать исходный код, генерируемый Razor, потому что это позволит проанализировать работу некоторых его функций.

Представления в приложении MVC не компилируются до запуска приложения, поэтому, чтобы увидеть классы, которые создаются Razor, необходимо запустить приложение и перейти по ссылке /Home/Index. Первый запрос к приложению MVC запускает процесс компиляции для всех представлений. Вывод для нашего запроса показан на рисунке 18-4.

**Рисунок 18-4:** Вывод метода действия Index контроллера Home



Для удобства классы, генерируемые из файлов представлений, записываются на диск в виде файлов кода C#, а затем компилируются, что значит, что вы можете увидеть операторы C# для представлений. В Windows 7 и 8 генерируемые файлы можно найти по адресу c:\Users\<yourLoginName>\AppData\Local\Temp\Temporary ASP.NET Files.

Найти код для конкретного представления будет немного сложнее. Как правило, здесь вы найдете несколько папок с загадочными именами, причем названия файлов .cs не будут соответствовать именам классов, которые в них содержатся. Например, мы обнаружили класс, созданный для представления из листинга 18-10, в файле App\_Web\_cuymyfa4.0.cs в папке root\bdd11980\ec057b05. Мы его отредактировали и сделали более удобным для чтения, и вы можете увидеть его в листинге 18-11.

**Листинг 18-11:** Класс C#, сгенерированный для представления Razor

```
namespace ASP
{
    using System;
    using System.Collections.Generic;
    using System.IO;
    using System.Linq;
    using System.Net;
    using System.Web;
    using System.Web.Helpers;
    using System.Web.Security;
    using System.Web.UI;
    using System.Web.WebPages;
    using System.Web.Mvc;
    using System.Web.Mvc.Ajax;
    using System.Web.Mvc.Html;
    using System.Web.Optimization;
    using System.Web.Routing;
    public class _Page_VIEWS_Home_Index_cshtml : System.Web.Mvc.WebViewPage<string[]>
    {
```

```
public _Page_VIEWS_Home_Index_cshtml()
{
}
public override void Execute()
{
    ViewBag.Title = "Index";
    WriteLiteral("\r\n\r\nThis is a list of fruit names:\r\n\r\n");
    foreach (string name in Model)
    {
        WriteLiteral(" <span><b>");
        Write(name);
        WriteLiteral("</b></span>\r\n");
    }
}
```

Во-первых, отметим, что класс наследует от `WebViewPage<T>`, где `T` – это тип модели: в нашем примере `WebViewPage<string[]>`. Так обрабатываются сильно типизированные представления. Также обратите внимание на имя класса: `_Page_VIEWS_Home_Index_cshtml`. Как видите, здесь закодирован путь к файлу представления. Таким образом Razor соотносит запросы к представлениям с экземплярами скомпилированных классов.

В методе `Execute` вы можете увидеть, как были обработаны операторы и элементы представления. Фрагменты кода, которые мы отметили префиксом `@`, записываются без изменений как операторы C#. Элементы HTML обрабатываются методом `WriteLiteral`, который записывает содержимое параметра в результат. В отличие от него, метод `Write` используется для переменных C# и кодирует значения строк, чтобы сделать их безопасными для использования в HTML-страницах.

Оба метода - `Write`, и `WriteLiteral` - записывают содержимое в объект `TextWriter`. Это тот же объект, который передается в метод `IView.Render`, с которым мы работали в начале этой главы. Цель скомпилированного представления Razor - сгенерировать статический и динамический контент и отправить его клиенту с помощью `TextWriter`. Имейте это в виду, когда мы будем рассматривать вспомогательные методы HTML далее в этой главе.

## Настраиваем адреса поиска представлений

При поиске представлений движок Razor следует соглашению, установленному в более ранних версиях MVC Framework. Например, если вы запрашиваете представление `Index`, связанное с контроллером `Home`, Razor просмотрит следующий список представлений:

- ~ / Views / Home / Index.cshtml;
  - ~ / Views / Home / Index.vbhtml;
  - ~ / Views / Shared / Index.cshtml;
  - ~ / Views / Shared / Index.vbhtml.

Как вы теперь знаете, Razor на самом деле не ищет файлы на диске, потому что они уже скомпилированы в классы C#. Razor ищет скомпилированный класс, который содержит код данных представлений. Файлы .cshtml представляют собой шаблоны, содержащие операторы C#, а файлы .vbhtml содержат операторы Visual Basic.

Чтобы изменить файлы представлений, которые ищет Razor, нужно создать подкласс `RazorViewEngine`. Он является реализацией `IViewEngine` для Razor. Он наследует ряд базовых классов, которые определяют набор свойств, указывающих, какие файлы представлений нужно искать. Эти свойства описаны в таблице 18-1.

**Таблица 18-1:** Свойства поиска движка Razor

Свойство	Описание	Значение по умолчанию
ViewLocationFormats MasterLocationFormats PartialViewLocationFormats	Адреса для поиска представлений, частичных представлений и макетов	~/Views/{1}/{0}.cshtml, ~/Views/{1}/{0}.vbhtml, ~/Views/Shared/{0}.cshtml, ~/Views/Shared/{0}.vbhtml
AreaViewLocationFormats AreaMasterLocationFormats AreaPartialViewLocationFormats	Адреса для поиска представлений, частичных представлений и макетов для областей	~/Areas/{2}/Views/{1}/{0}.cshtml, ~/Areas/{2}/Views/{1}/{0}.vbhtml, ~/Areas/{2}/Views/Shared/{0}.cshtml, ~/Areas/{2}/Views/Shared/{0}.vbhtml

Эти свойства предшествовали появлению Razor, поэтому каждой группе из трех свойств соответствует один набор значений. Каждый набор значений представляет собой массив строк. Ниже приведены значения параметров, которые соответствуют заполнителям:

- {0} - имя представления.
- {1} - имя контроллера.
- {2} - имя области.

Для изменения адресов поиска создайте новый класс, наследующий от `RazorViewEngine`, и измените значения для одного или нескольких свойств, описанных в таблице 18-1.

Чтобы продемонстрировать, как изменять адреса поиска, мы добавили в приложение проект `Infrastructure` и создали движок представлений под названием `CustomLocationViewEngine`, который показан в листинге 18-12.

**Листинг 18-12:** Изменяем адреса поиска в Razor

```
using System.Web.Mvc;

namespace WorkingWithRazor.Infrastructure
{
    public class CustomLocationViewEngine : RazorViewEngine
    {
        public CustomLocationViewEngine()
        {
            ViewLocationFormats = new string[] {"~/Views/{1}/{0}.cshtml",
"~/Views/Common/{0}.cshtml"};
        }
    }
}
```

Мы установили новое значение для `ViewLocationFormats`. Наш новый массив содержит записи только для файлов `.cshtml`. Кроме того, мы изменили каталог для общих представлений на `Views/Common` вместо `Views/Shared`. Далее мы зарегистрировали наш пользовательский движок, используя коллекцию `ViewEngines.Engines` в методе `Application_Start` файла `Global.asax`, как показано в листинге 18-13.

### Листинг 18-13: Регистрируем пользовательский движок представлений

```
using System.Web.Http;
using System.Web.Mvc;
using System.Web.Optimization;
using System.Web.Routing;
using WorkingWithRazor.Infrastructure;

namespace WorkingWithRazor
{
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();

            ViewEngines.Engines.Clear();
            ViewEngines.Engines.Add(new CustomLocationViewEngine());

            WebApiConfig.Register(GlobalConfiguration.Configuration);
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);
        }
    }
}
```

Вы должны помнить, что механизм вызова действий обращается к каждому движку по очереди, проверяя, какой из них может найти представление. Когда нам нужно было добавить в коллекцию пользовательский движок, она уже содержала стандартный движок Razor. Чтобы избежать конкуренции с этой реализацией, мы вызвали метод `Clear` и удалили все зарегистрированные движки, а затем зарегистрировали наш пользовательский движок с помощью метода `Add`.

Чтобы продемонстрировать изменение адресов поиска, мы создали папку `/Views/Common` и добавили в нее файл представления под названием `List.cshtml`. Содержимое этого файла показано в листинге 18-14.

### Листинг 18-14: Содержимое файла `/Views/Common/List.cshtml`

```
@{
    ViewBag.Title = "List";
}
<h3>This is the /Views/Common/List.cshtml View</h3>
```

Для отображения этого представления мы добавили в контроллер `Home` метод действия, показанный в листинге 18-15.

### Листинг 18-15: Добавляем метод действия в контроллер `Home`

```
using System.Web.Mvc;

namespace WorkingWithRazor.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            string[] names = {"Apple", "Orange", "Pear"};
            return View(names);
        }

        public ActionResult List()
        {

```

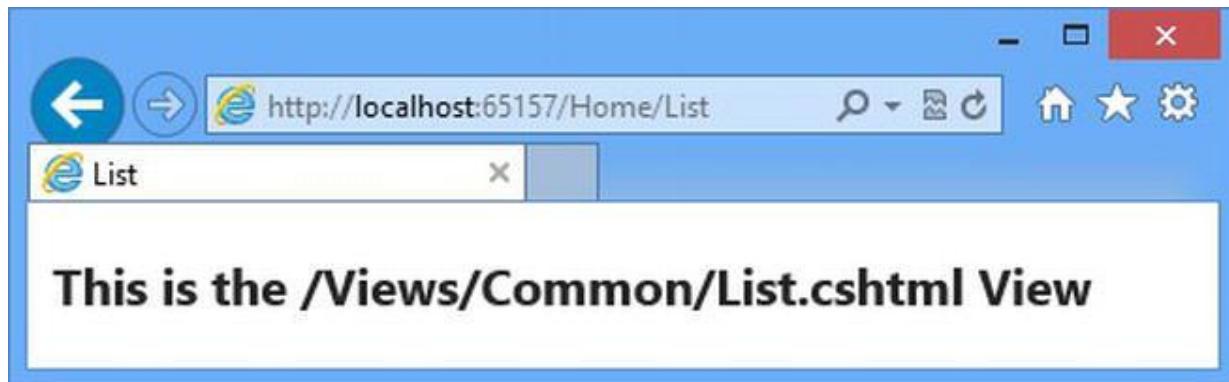
```

        return View();
    }
}
}

```

Если мы запустим приложение и перейдем по ссылке /Home/List, поиск файла представления List.cshtml будет осуществляться в добавленной нами папке Views/Common, как показано на рисунке 18-5.

**Рисунок 18-5:** Результат добавления новых адресов поиска в движок представления



## Добавление динамического контента в представление Razor

Главная цель представлений - визуализировать компоненты доменной модели как компоненты пользовательского интерфейса. Для этого нужно уметь добавлять в представления *динамический контент*. Динамический контент генерируется во время выполнения и может быть разным для разных запросов. Это его отличие от *статического контента*, такого как HTML, который вы создаете при написании приложения и который остается одинаковым для всех запросов.

Динамический контент можно добавить в представления несколькими способами, которые описаны в таблице 18-2.

**Таблица 18-2:** Способы добавления динамического контента в представления

Способ	Используется для
Код	Используется для создания небольших, независимых частей логики представления, например, операторы <code>if</code> или <code>foreach</code> . Это основной способ создания динамического контента в представлении, и на нем основаны некоторые другие подходы. Мы рассмотрели его в главе 5 и уже использовали во многих примерах в других главах.
Вспомогательные методы HTML	Используются для создания одного элемента HTML или небольшой коллекции элементов, обычно на основании данных модели представления или объекта <code>ViewData</code> . Можно использовать встроенные вспомогательные методы MVC Framework, можно также создавать свои собственные. Вспомогательные методы HTML рассматриваются в главе 19.
Секции	Используются для разбиения контента на блоки, которые будут вставлены в макет в

Способ	Используется для
	определенных местах.
Частичные представления	Используются для включения подсекции разметки в несколько представлений. Частичные представления могут содержать код, вспомогательные методы HTML и ссылки на другие частичные представления. Частичные представления не могут вызывать методы действий, поэтому их нельзя использовать для выполнения бизнес-логики.
Дочерние действия	Используются для создания повторно используемых элементов управления UI и виджетов, в которых необходима бизнес-логика. Дочернее действие вызывает метод действия, визуализирует представление и внедряет результат в поток ответа.

Первые два способа рассматриваются в других главах этой книги, а три последних мы опишем в последующих разделах.

## Используем секции

Движок Razor поддерживает концепцию секций, которые позволяют выделять различные области в макете. Секции Razor позволяют разбивать представление на части и контролировать то, где они внедряются в макет. Чтобы продемонстрировать работу секций, мы отредактировали файл /Views/Home/Index.cshtml, как показано в листинге 18-16.

**Листинг 18-16:** Определяем секцию в представлении

```
@model string[]
@{
    ViewBag.Title = "Index";
}

@section Header {
    <div class="view">
        @foreach (string str in new[] { "Home", "List", "Edit" })
        {
            @Html.ActionLink(str, str, null, new { style = "margin: 5px" })
        }
    </div>
}

<div class="view">
    This is a list of fruit names:
    @foreach (string name in Model)
    {
        <span><b>@name</b></span>
    }
</div>

@section Footer {
    <div class="view">
        This is the footer
    </div>
}
```

Секции определяются с помощью тега Razor @, за которым следует имя секции. В нашем примере их две - Header и Footer. Их содержание представляет собой обычную смесь разметки HTML и тегов Razor.

Чтобы указать, где в макете должны отображаться секции, используется вспомогательный метод `@RenderSection`. В листинге 18-17 показаны изменения, которые мы внесли в файл `~/Views/Shared/_Layout.cshtml`.

Подсказка

*Движок представлений все еще использует наши пользовательские адреса поиска, что означает, что макеты находятся в папке /views/Shared, а представления - в /Views/Common.*

---

**Листинг 18-17:** Используем секции в макете

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <style type="text/css">
        div.layout {
            background-color: lightgray;
        }

        div.view {
            border: thin solid black;
            margin: 10px 0;
        }
    </style>
    <title>@ViewBag.Title</title>
</head>
<body>
    @RenderSection("Header")

    <div class="layout">
        This is part of the layout
    </div>

    @RenderBody()

    <div class="layout">
        This is part of the layout
    </div>

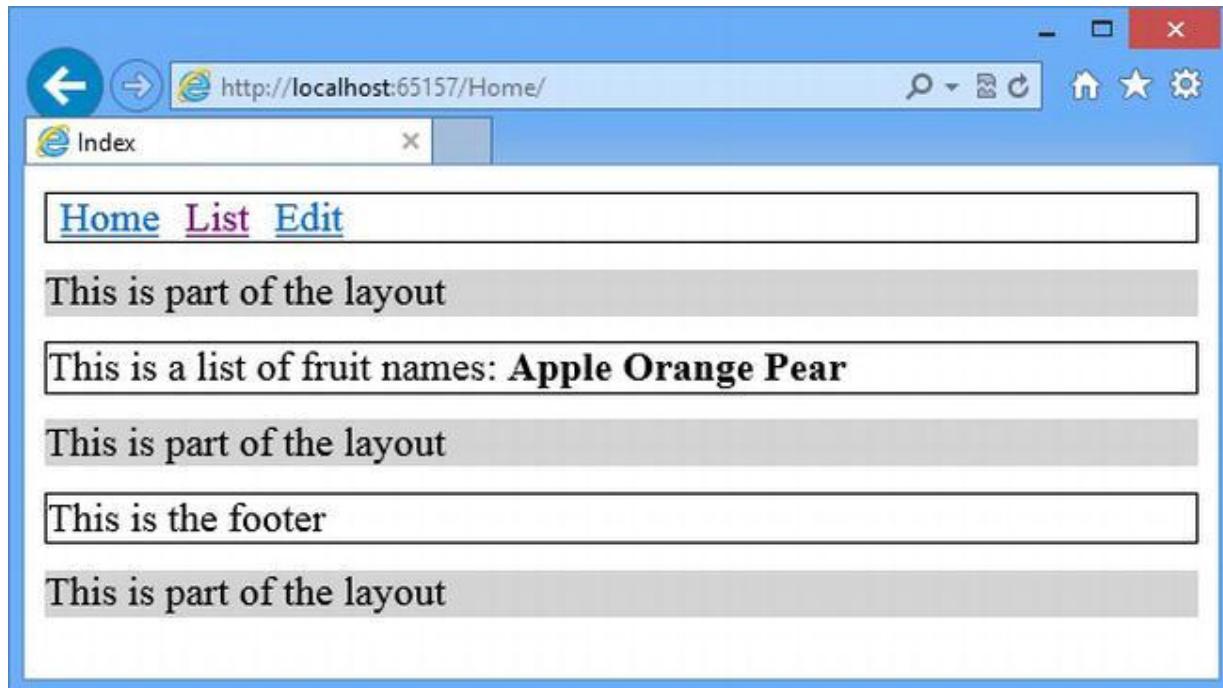
    @RenderSection("Footer")

    <div class="layout">
        This is part of the layout
    </div>
</body>
</html>
```

Когда Razor анализирует макет, он заменяет вспомогательный метод `RenderSection` на содержимое секции представления с указанным именем. Те части представления, которые не содержатся в секциях, вставляются в макет с помощью вспомогательного метода `RenderBody`.

Чтобы увидеть работу секций, запустите приложение, как показано на рисунке 18-6. Мы добавили некоторые базовые стили CSS, чтобы было понятно, какие из выведенных секций принадлежат представлению и какие – макету. Результат выглядит незамысловато, но он демонстрирует, как можно внедрить части контента представления в макет в заданной последовательности.

**Рисунок 18-6:** Используем секции в представлении, чтобы внедрить контент в макет



*Примечание*

*В представлении можно определить только те секции, на которые есть ссылки в макете. При попытке определить в представлении секции, для которых нет соответствующего вызова вспомогательного метода @RenderSection в макете, MVC Framework выбросит исключение.*

Смешивание секций с остальным кодом представления – нестандартный подход. По соглашению секции определяются либо в начале, либо в конце представления, чтобы легче было увидеть, какие области контента будет рассматриваться как секции и будут захвачены вспомогательным методом RenderBody. Мы часто используем и другой подход, согласно которому представление должно содержать только секции, тело представления также заключается в секцию, как показано в листинге 18-18.

**Листинг 18-18:** Определяем представление с помощью секций Razor

```
@model string[]
{
    ViewBag.Title = "Index";
}

@section Header {
    <div class="view">
        @foreach (string str in new[] { "Home", "List", "Edit" })
        {
            @Html.ActionLink(str, str, null, new { style = "margin: 5px" })
        }
    </div>
}
```

```

        </div>
    }

@section Body {
    <div class="view">
        This is a list of fruit names:
        @foreach (string name in Model)
        {
            <span><b>@name</b></span>
        }
    </div>
}

@section Footer {
    <div class="view">
        This is the footer
    </div>
}

```

Мы считаем, что такой подход позволяет создавать более понятные представления и снижает вероятность того, что RenderBody захватит посторонний контент. Чтобы его применить, мы должны заменить вызов вспомогательного метода RenderBody на RenderSection("Body"), как показано в листинге 18-19.

**Листинг 18-19:** Определяем представление с помощью RenderSection("Body")

```

<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <style type="text/css">
        div.layout {
            background-color: lightgray;
        }

        div.view {
            border: thin solid black;
            margin: 10px 0;
        }
    </style>
    <title>@ViewBag.Title</title>
</head>
<body>
    @RenderSection("Header")

    <div class="layout">
        This is part of the layout
    </div>

    @RenderSection("Body")

    <div class="layout">
        This is part of the layout
    </div>

    @RenderSection("Footer")

    <div class="layout">
        This is part of the layout
    </div>
</body>
</html>

```

## Тестирование секций

Вы можете проверить, определена ли в представлении какая-либо секция из макета. Это делается для того, чтобы предоставить контент по умолчанию для этой секции, если она отсутствует в представлении и мы по каким-то причинам не хотим ее там определять. Мы изменили файл \_Layout.cshtml, чтобы проверить, определена ли в нем секция Footer, как показано в листинге 18-20.

**Листинг 18-20:** Проверяем наличие секции Footer в представлении

```
@if (IsSectionDefined("Footer"))
{
    @RenderSection("Footer")
}
else
{
    <h4>This is the default footer</h4>
}
```

Вспомогательный метод IsSectionDefined возвращает true для заданного имени секции, если в текущем представлении определена эта секция. В этом примере с помощью вспомогательного метода мы выясняем, нужно ли предоставлять какой-либо контент по умолчанию, если окажется, что в представлении не определена секция Footer.

## Визуализируем дополнительные секции

По умолчанию в представлении должны быть определены все секции, к которым в макете имеются вызовы RenderSection. Если секции отсутствуют, MVC Framework покажет пользователю исключение. Чтобы продемонстрировать это, мы добавили в файл \_Layout.cshtml новый вызов RenderSection к секции под названием scripts, как показано в листинге 18-21 (это секция, которую Visual Studio добавляет в макет по умолчанию для проектов на шаблоне Basic, но которую мы удалили в самом начале).

**Листинг 18-21:** Добавляем в макет вызов RenderSection, для которого нет соответствующей секции в представлении

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <style type="text/css">
        div.layout {
            background-color: lightgray;
        }

        div.view {
            border: thin solid black;
            margin: 10px 0;
        }
    </style>
    <title>@ViewBag.Title</title>
</head>
<body>
    @RenderSection("Header")

    <div class="layout">
        This is part of the layout
    </div>
```

```

@RenderSection("Body")

<div class="layout">
    This is part of the layout
</div>

@if (IsSectionDefined("Footer"))
{
    @RenderSection("Footer")
}
else
{
    <h4>This is the default footer</h4>
}

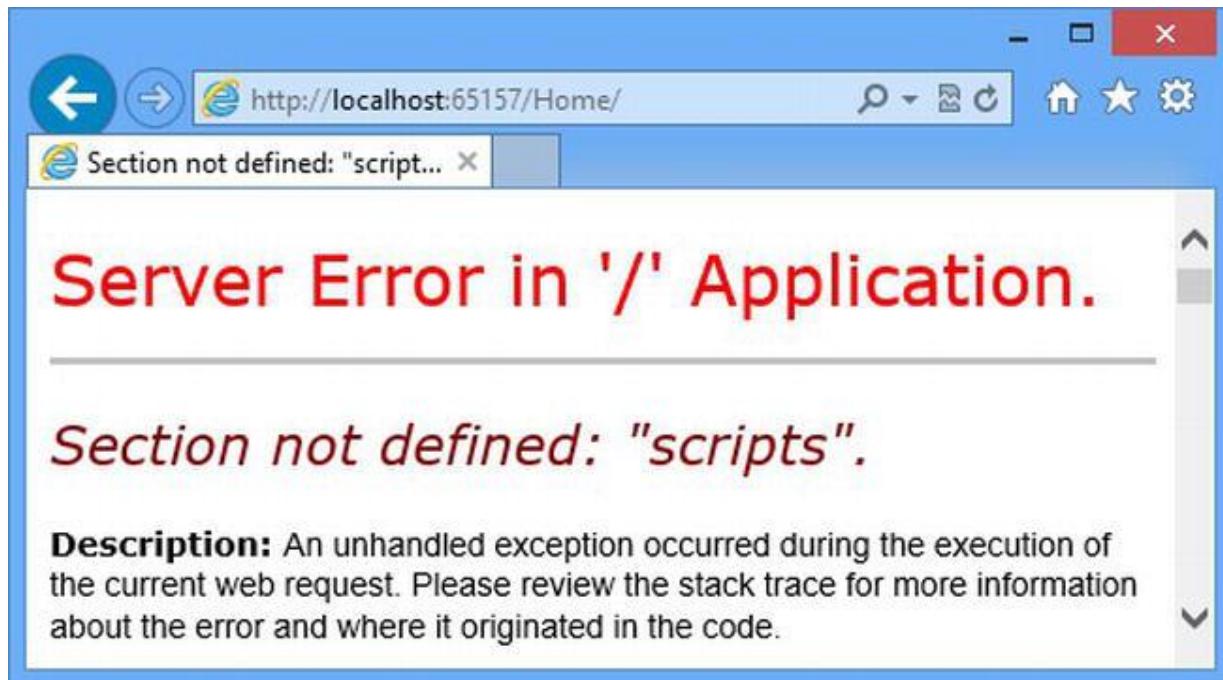
@RenderSection("scripts")

<div class="layout">
    This is part of the layout
</div>
</body>
</html>

```

Если мы запустим приложение и движок Razor попытается визуализировать макет и представление, то увидим ошибку, показанную на рисунке 18-7.

**Рисунок 18-7:** Ошибка, которая появляется в случае отсутствия секции



Вы можете использовать метод `IsSectionDefined`, чтобы избежать вызовов `RenderSection` к секциям, не определенным в представлении, но для этого есть более элегантный подход: передавать дополнительное значение `false` в метод `RenderSection`, как показано в листинге 18-22.

**Листинг 18-22:** Делаем секцию необязательной

```
@RenderSection("scripts", false)
```

Это делает секцию необязательной: если она определена в представлении, ее содержание будет вставлено в результат, если нет – мы не получим исключение.

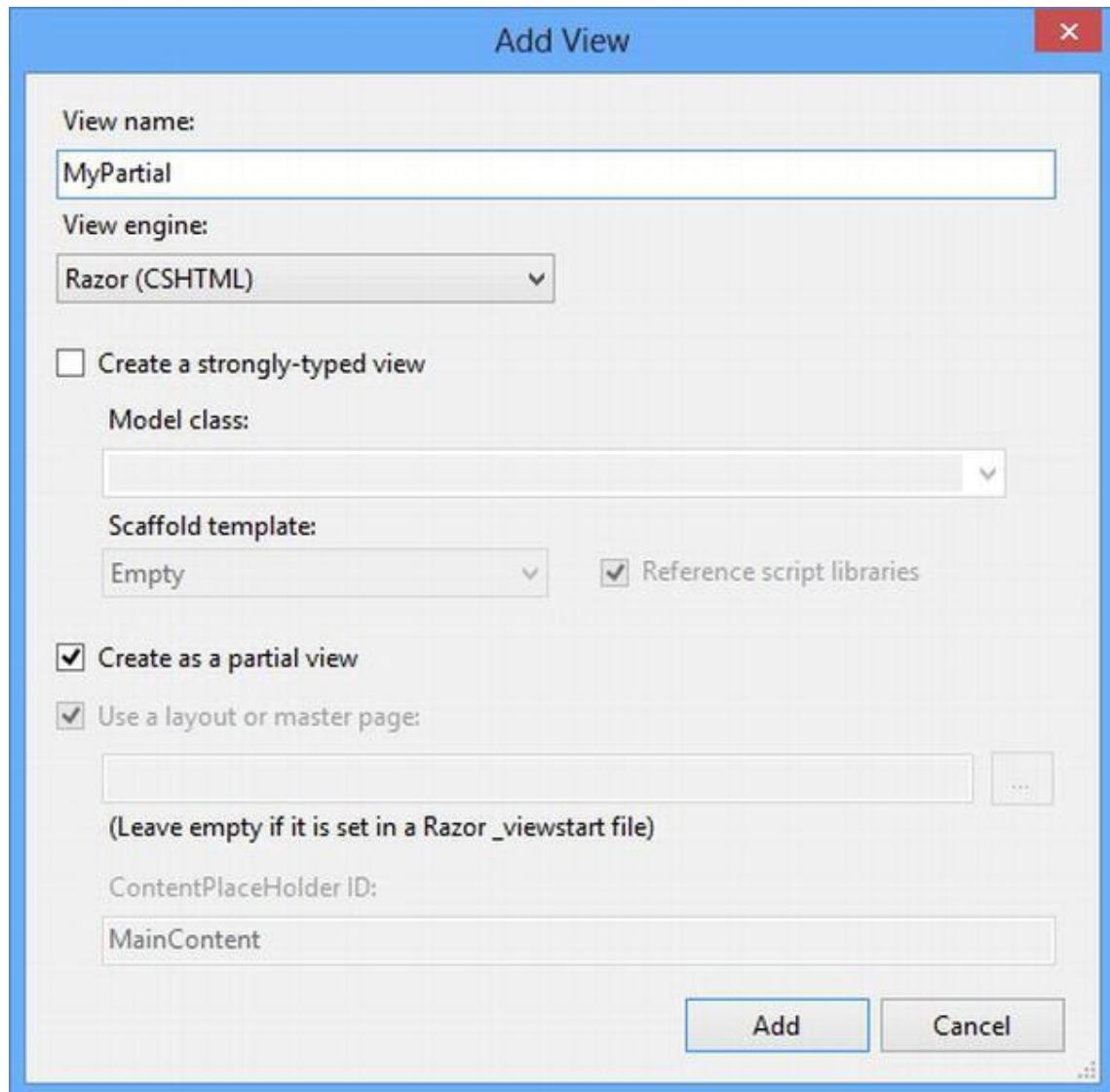
## Используем частичные представления

В приложениях мы часто используем одни и те же фрагменты тегов Razor и HTML-разметки в нескольких представлениях. Чтобы не дублировать контент, можно использовать *частичные представления*. Они представляют собой отдельные файлы, которые содержат фрагменты кода с тегами и разметкой и могут быть включены в другие представления. В этом разделе мы покажем вам, как создавать и использовать частичные представления, объясним принципы их работы и продемонстрируем способы передачи в них данных представлений.

### Создаем частичное представление

Для начала создадим частичное представление под названием `MyPartial`. Для этого кликните правой кнопкой мыши по папке `/Views/Shared`, выберите из контекстного меню `Add - View` и отметьте флажком опцию `Create as Partial View`, как показано на рисунке 18-8.

**Рисунок 18-8:** Создаем частичное представление



Сразу после создания оно будет пустым, и мы добавим в него содержимое, показанное в листинге 18-23.

**Листинг 18-23:** Файл MyPartial.cshtml

```
<div>
    This is the message from the partial view.
    @Html.ActionLink("This is a link to the Index action", "Index")
</div>
```

Мы хотим продемонстрировать, как можно смешивать HTML-разметку и теги Razor, поэтому мы определили в нем простое сообщение и вызов к вспомогательному методу ActionLink.

Чтобы использовать частичное представление, нужно вызвать вспомогательный метод HTML Partial из другого представления. Для демонстрации мы внесли изменения в файл ~/Views/Common/List.cshtml, как показано в листинге 18-24.

**Листинг 18-24:** Используем частичное представление

```
@{
    ViewBag.Title = "List";
    Layout = null;
}

<h3>This is the /Views/Common/List.cshtml View</h3>

@Html.Partial("MyPartial")
```

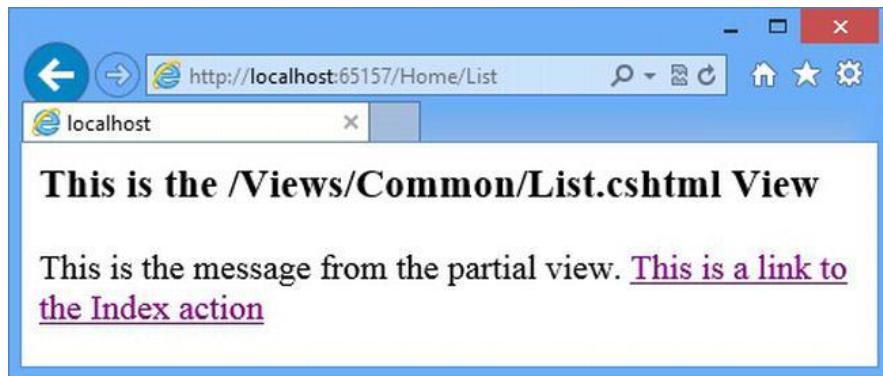
Имя файла представления указывается без расширения. Движок представлений будет искать указанное частичное представление по обычным адресам поиска, то есть при визуализации представления для контроллера Home - в папках /Views/Home и /Views/Shared. (Мы установили переменной Layout значение null, так что теперь нам не нужно указывать секции, которые использовались ранее в этой главе.)

*Подсказка*

*Движок представлений Razor ищет частичные представления так же, как и обычные (в папках ~/views/<controller> и ~/views/Shared). Это значит, что можно создавать специализированные версии частичных представлений для конкретных контроллеров и переопределять частичные представления с одинаковыми именами в папке Shared. Хотя последняя функция может показаться странной, она очень удобна, потому что чаще всего частичные представления используются для визуализации контента в макете.*

Вы можете увидеть вывод частичного представления, запустив приложение и перейдя по ссылке /Home>List, как показано на рисунке 18-9.

**Рисунок 18-9:** Вывод частичного представления



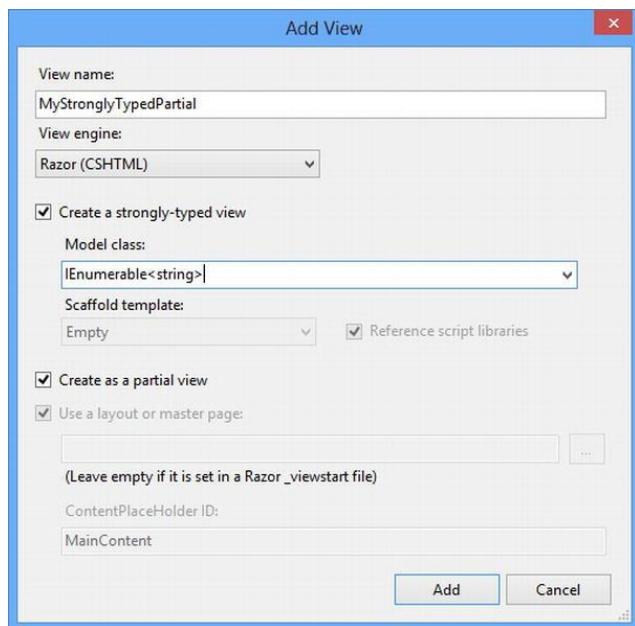
#### Подсказка

С помощью вызова вспомогательного метода `ActionLink` в частичном представлении мы получаем из обрабатываемого запроса информацию о контроллере. Это значит, что если мы указываем метод `Index`, элемент `a` будет относиться к контроллеру `Home`, так как именно этот контроллер вызывает визуализацию частичного представления. Если мы используем частичные представления в представлении, которое визуализируется другим контроллером, то `ActionLink` сгенерирует ссылку на этот контроллер. Мы вернемся к вспомогательным методам HTML в главе 19.

### Используем строго типизированные частичные представления

Можно также создать строго типизированное частичное представление, а затем передавать в него объекты моделей представлений, которые оно будет визуализировать. Чтобы это продемонстрировать, мы создали новое строго типизированное частичное представление под названием `MyStronglyTypedPartial.cshtml` в папке `/Views/Shared`. Оно создается практически так же, как и обычное частичное представление, только нужно выбрать или ввести с клавиатуры тип для опции `Model class`, как показано на рисунке 18-10. Мы использовали тип `IEnumerable<string>`.

**Рисунок 18-10:** Создаем строго типизированное частичное представление



В листинге 18-25 показаны дополнения в файле частичного представления, который был создан Visual Studio. Изначально он содержит только тег модели @, который указывает тип модели представления.

**Листинг 18-25:** Создаем строго типизированное частичное представление

```
@model IEnumerable<string>

<div>
    This is the message from the partial view.
    <ul>
        @foreach (string str in Model)
        {
            <li>@str</li>
        }
    </ul>
</div>
```

Мы отображаем содержимое объекта модели представления как список HTML с помощью тега Razor @foreach. Чтобы продемонстрировать работу этого частичного представления, мы обновили файл /Views/Common/List.cshtml, как показано в листинге 18-26.

**Листинг 18-26:** Используем строго типизированное частичное представление

```
@{
    ViewBag.Title = "List";
    Layout = null;
}

<h3>This is the /Views/Common/List.cshtml View</h3>

@Html.Partial("MyStronglyTypedPartial", new[] { "Apple", "Orange", "Pear" })
```

Отличие от предыдущего примера заключается в том, что здесь мы передаем дополнительный аргумент во вспомогательный метод Partial, который определяет объект модели представления. Чтобы увидеть, как работает это строго типизированное частичное представление, запустите приложение и перейдите по ссылке /Home>List, как показано на рисунке 18-11.

**Рисунок 18-11:** Используем строго типизированное частичное представление



## Используем дочерние действия

Дочерние действия – это методы действий, которые вызываются из представления. Они позволяют избежать дублирования логики контроллера, которую необходимо использовать в приложении несколько раз. Дочерние действия так же относятся к действиям, как частичные представления – к представлениям.

Дочерние действия чаще всего используются для отображения какого-либо управляемого данными виджета, который должен появляться на нескольких страницах и содержит данные, не относящиеся к основному действию. Мы их использовали в примере приложения SportsStore, когда необходимо было во все страницы включить управляемое данными меню навигации, без необходимости поставлять данные о категориях непосредственно от каждого действия метода. Эти данные поставлялись независимо дочерним действием.

### Создаем дочернее действие

Любое действие можно использовать как дочернее. Чтобы продемонстрировать работу дочерних действий, мы добавили новый метод действия в контроллер Home, как показано в листинге 18-27.

**Листинг 18-27:** Добавляем дочернее действие в контроллер Home

```
using System.Web.Mvc;
using System;

namespace WorkingWithRazor.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            string[] names = {"Apple", "Orange", "Pear"};
            return View(names);
        }

        public ActionResult List()
        {
            return View();
        }

        [ChildActionOnly]
        public ActionResult Time()
        {
            return PartialView(DateTime.Now);
        }
    }
}
```

Наш метод действия называется Time и визуализирует частичное представление, вызывая метод PartialView (о котором мы рассказывали в главе 15). Атрибут ChildActionOnly гарантирует, что метод действия может быть вызван только как дочернее действие из представления. Хотя метод действия может использоваться как дочерний и без этого атрибута, мы всегда применяем ChildActionOnly, чтобы данный метод невозможно было вызвать на запрос пользователя.

После определения метода действия нам нужно создать частичное представление, которое он будет визуализировать. Дочерние действия обычно связываются с частичными представлениями, хотя это и необязательно. В листинге 18-28 показано представление /Views/Home/Time.cshtml, которое мы создали для работы с нашим дочерним действием. Это строго типизированное частичное представление, моделью которого является объект DateTime.

**Листинг 18-28:** Частичное представление для дочернего действия

```
@model DateTime  


The time is: @Model.ToShortTimeString()


```

**Визуализируем дочернее действие**

Дочернее действие вызывается вспомогательным методом `Html.Action`. После данного вызова выполняется метод действия, обрабатывается `ViewResult`, и вывод внедряется в ответ клиенту. В листинге 18-29 показаны изменения, которые мы внесли в файл `/Views/Common/List.cshtml`, чтобы визуализировать дочернее действие.

**Листинг 18-29:** Вызов дочернего действия из представления `List`

```
@{  
    ViewBag.Title = "List";  
    Layout = null;  
}  
  
<h3>This is the /Views/Common/List.cshtml View</h3>  
  
@Html.Partial("MyStronglyTypedPartial", new[] { "Apple", "Orange", "Pear" })  
  
@Html.Action("Time")
```

Вы можете увидеть вывод дочернего действия, запустив приложение и снова перейдя по ссылке `/Home/List`, как показано на рисунке 18-12.

**Рисунок 18-12:** Используем дочернее действие



Вспомогательному методу `Action`, который был вызван в листинге 18-29, мы передали параметр с именем вызываемого метода действия. MVC Framework будет искать метод действия в контроллере,

который обрабатывает текущий запрос. Чтобы вызвать метод действия из другого контроллера, передайте в параметр его имя, например:

```
@Html.Action("Time", "MyController")
```

В методы действий можно передавать параметры с помощью анонимно типизированных объектов, свойства которых соответствуют именам параметров дочернего метода действия. Так, например, следующее дочернее действие:

```
[ChildActionOnly]
public ActionResult Time(DateTime time)
{
    return PartialView(time);
}
```

можно вызвать из представления следующим образом:

```
@Html.Action("Time", new { time = DateTime.Now })
```

## Резюме

В этой главе мы исследовали систему представлений MVC и движок представлений Razor. Вы научились создавать собственные движки представлений, настраивать поведение стандартного движка Razor и познакомились со всеми доступными техниками добавления динамического контента в представление. Следующая глава будет посвящена вспомогательным методам, с помощью которых можно генерировать контент для использования в представлениях.

# Вспомогательные методы

В этой главе мы рассмотрим вспомогательные методы, которые содержат фрагменты кода и разметки для повторного использования в приложении MVC. Для начала мы научимся создавать собственные вспомогательные методы. Также в этой и следующих главах мы изучим многочисленные встроенные вспомогательные методы, начиная с методов для создания элементов HTML `form`, `input` и `select`.

## Создание проекта для примера

Для этой главы мы создали новый проект Visual Studio под названием `HelperMethods`, используя шаблон `Basic`. В него был добавлен контроллер `Home`, который показан в листинге 19-1.

**Листинг 19-1:** Контроллер `Home` в проекте `HelperMethods`

```
using System.Web.Mvc;

namespace HelperMethods.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewBag.Fruits = new string[] {"Apple", "Orange", "Pear"};
            ViewBag.Cities = new string[] {"New York", "London", "Paris"};

            string message = "This is an HTML element: <input>";

            return View((object) message);
        }
    }
}
```

Мы передаем в представление пару строковых массивов через `ViewBag` и устанавливаем `string` в качестве объекта модели. Мы создали в папке `/Views/Home` файл представления `Index.cshtml`, содержимое которого вы можете увидеть в листинге 19-2. Это строго типизированное представление (типов модели является `string`), для которого не используется макет.

**Листинг 19-2:** Содержимое файла `Index.cshtml`

```
@model string
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
        Here are the fruits:
        @foreach (string str in (string[])ViewBag.Fruits)
        {
            <b>@str </b>
        }
    </div>
</div>
```

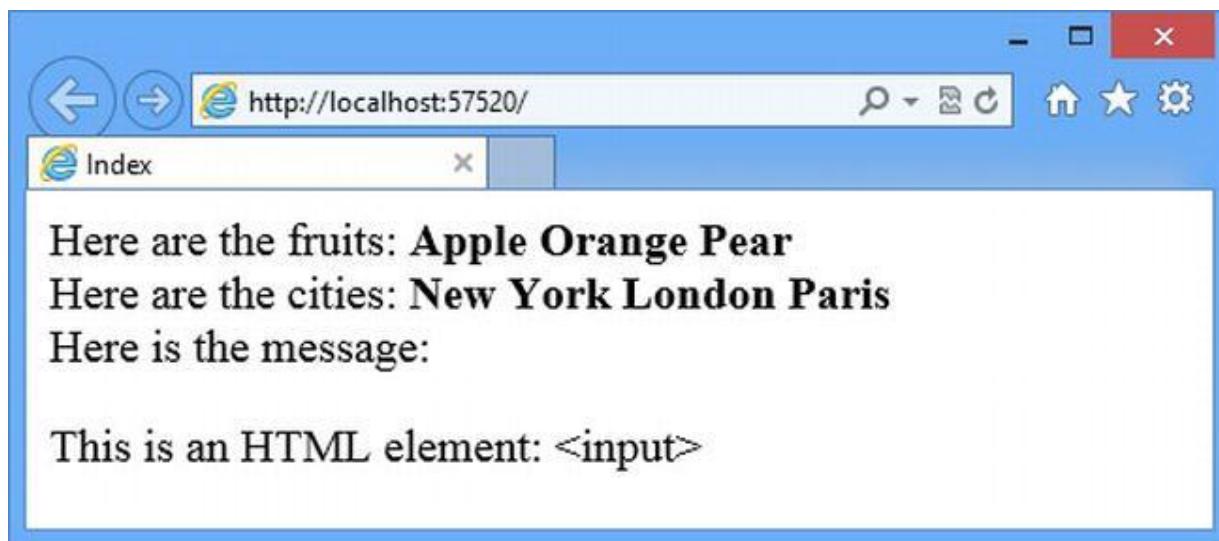
```

Here are the cities:
@foreach (string str in (string[])ViewBag.Cities)
{
    <b>@str </b>
}
</div>
<div>
    Here is the message:
    <p>@Model</p>
</div>
</body>
</html>

```

Чтобы увидеть, как визуализируется представление, запустите приложение. В соответствии со стандартной конфигурацией маршрутизации, которую добавляет в проект Visual Studio, корневой URL, автоматически запрошенный браузером, будет соотнесен с действием Index контроллера Home, как показано на рисунке 19-1.

**Рисунок 19-1:** Запускаем приложение



## Создание пользовательских вспомогательных методов

Следуя плану, установившемуся в последних нескольких главах, мы начнем изучение вспомогательных методов с создания собственной реализации. В следующих разделах мы продемонстрируем две техники для создания пользовательских вспомогательных методов.

### Создаем внутренний вспомогательный метод

Самый простой вид вспомогательных методов - внутренние (*inline*) вспомогательные методы, которые определяются внутри представления. Чтобы упростить наш пример представления, мы можем создать внутренний вспомогательный метод с помощью тега @helper, как показано в листинге 19-3.

**Листинг 19-3:** Создаем внутренний вспомогательный метод

```
@model string
```

```
@{
```

```

        Layout = null;
    }

@helper ListArrayItems(string[] items)
{
    foreach (string str in items)
    {
        <b>@str </b>
    }
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
        Here are the fruits: @ListArrayItems(ViewBag.Fruits)
    </div>
    <div>
        Here are the cities: @ListArrayItems(ViewBag.Cities)
    </div>
    <div>
        Here is the message:
        <p>@Model</p>
    </div>
</body>
</html>

```

Внутренние вспомогательные методы имеют такие же имена и параметры, как и обычные методы C#. В данном примере мы определили вспомогательный метод под названием `ListArrayItems`, который принимает в качестве параметра строковый массив. Хотя внутренний вспомогательный метод выглядит как метод, он не возвращает никакого значения. Его содержание обрабатывается и помещается в ответ клиенту.

#### **Подсказка**

*Обратите внимание, что при использовании внутренних вспомогательных методов не нужно приводить тип динамических свойств из `ViewBag` к строковым массивам.*

*Одной из особенностей этого вида вспомогательных методов является то, что он может приводить типы во время выполнения.*

---

Тело внутреннего вспомогательного метод имеет такой же синтаксис, как и все остальное представление Razor. Строки литералов рассматриваются как статический HTML, а операторы, которые требуют обработки Razor, отмечаются префиксом `@`. Вспомогательный метод в примере смешивает статический HTML и теги Razor для перечисления элементов в массиве и дает тот же результат, что и наше первоначальное представление, сокращая дублирование кода и разметки.

Преимущество этого подхода заключается в том, что для изменения способа отображения содержимого массива нам нужно сделать только одно изменение. Например, в листинге 19-4 показано, как мы заменили обычную запись значений из массива на запись в виде ненумерованного списка HTML.

#### **Листинг 19-4:** Изменяем содержимое вспомогательного метода

```

@helper ListArrayItems(string[] items)
{

```

```

<ul>
    @foreach (string str in items)
    {
        <li>@str</li>
    }
</ul>
}

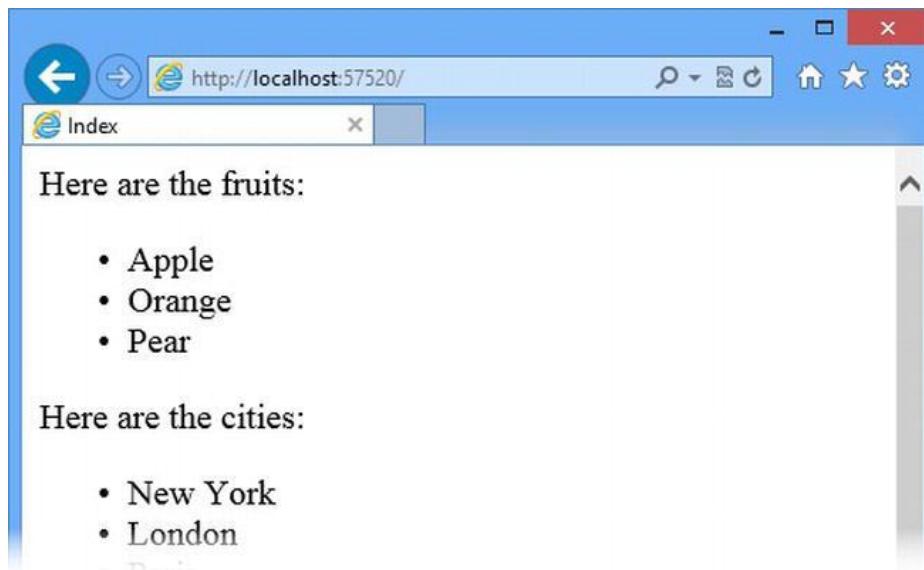
```

Нужно внести только одно изменение - это может показаться незначительным преимуществом в таком простом примере, но в реальных проектах поможет сохранить представления простыми и последовательными. Результат этого изменения показан на рисунке 19-2.

#### *Подсказка*

*Обратите внимание, что в этом примере мы отметили префиксом @ ключевое слово foreach, а в листинге 19-3 - нет. Дело в том, что первый элемент в теле вспомогательного метода изменился на элемент HTML, следовательно, мы должны сообщить Razor с помощью тега @, что мы используем оператор C#. В предыдущем примере элементов HTML не было, так что Razor рассматривал содержание тела вспомогательного метода как код. Такие мелочи было бы трудно отслеживать, но, к счастью, Visual Studio их подсвечивает.*

**Рисунок 19-2:** Измененяем разметку в вспомогательном методе



#### **Создаем внешние вспомогательные методы**

Внутренние вспомогательные методы удобны, но их можно использовать только в том представлении, в котором они были объявлены, и из-за своей сложности или объема могут затруднить его чтение.

В качестве альтернативы можно использовать внешние (*external*) вспомогательные методы HTML, которые выражаются как методы расширения C#. Внешние вспомогательные методы используются шире, но и создавать их сложнее, потому что C# не очень хорошо обрабатывает элементы HTML.

Чтобы продемонстрировать их работу, мы создали в проекте папку Infrastructure, а в ней - новый класс CustomHelpers.cs. Содержимое этого файла показано в листинге 19-5.

**Листинг 19-5:** Содержимое файла CustomerHelpers.cs

```
using System.Web.Mvc;

namespace HelperMethods.Infrastructure
{
    public static class CustomHelpers
    {
        public static MvcHtmlString ListArrayItems(this HtmlHelper html, string[] list)
        {
            TagBuilder tag = new TagBuilder("ul");
            foreach (string str in list)
            {
                TagBuilder itemTag = new TagBuilder("li");
                itemTag.SetInnerText(str);
                tag.InnerHtml += itemTag.ToString();
            }
            return new MvcHtmlString(tag.ToString());
        }
    }
}
```

Данный вспомогательный метод выполняет ту же функцию, что и встроенный вспомогательный метод в предыдущем примере - он принимает массив строк и генерирует HTML-элемент `ul`, содержащий элемент `li` для каждой строки в массиве.

Первый параметр этого вспомогательного метода – объект `HtmlHelper` с ключевым словом `this`, которое сообщает компилятору C#, что мы определяем метод расширения. Свойства `HtmlHelper` обеспечивают доступ к информации, которая может быть полезна при создании контента; они описаны в таблице 19-1.

**Таблица 19-1:** Свойства класса `HtmlHelper`

Свойство	Описание
<code>RouteCollection</code>	Возвращает набор маршрутов, определенных в приложении.
<code>ViewBag</code>	Возвращает данные объекта из <code>ViewBag</code> , который был передан методом действия в представление, вызвавшее данный вспомогательный метод.
<code>ViewContext</code>	Возвращает объект <code>ViewContext</code> , который обеспечивает доступ к информации о запросе и процессе его обработки (и который мы опишем далее в этой главе).

Свойство `ViewContext` наиболее полезно для создания контента, который адаптируется к текущему запросу. В таблице 19-2 описаны наиболее часто используемые свойства, определенные классом `ViewContext`.

**Таблица 19-2:** Свойства класса `ViewContext`

Свойство	Описание
<code>Controller</code>	Возвращает контроллер, обрабатывающий текущий запрос.
<code>HttpContext</code>	Возвращает объект <code>HttpContext</code> для текущего запроса.
<code>IsChildAction</code>	Возвращает <code>true</code> , если вызвавшее вспомогательный метод представление визуализируется дочерним действием (дочерние действия рассмотрены в главе 18).
<code>RouteData</code>	Возвращает данные маршрутизации для запроса.
<code>View</code>	Возвращает экземпляр реализации <code>IView</code> , которая вызвала вспомогательный метод.

С помощью вспомогательных методов вы можете получить всеобъемлющую информацию о запросе, но по большей части они используются, чтобы сохранить последовательность форматирования.

Чтобы генерировать контент для определенных запросов, можно использовать встроенные (*built-in*) вспомогательные методы (они будут описаны далее в этой главе) или с помощью частичных представлений и дочерних действий реализовывать более сложные решения.

В нашем примере вспомогательного метода не требуется никакой информации о запросе, но нужно создать несколько элементов HTML. Самый простой способ создать код HTML во вспомогательных методах - это использовать класс `TagBuilder`, который позволяет создавать строки HTML, не используя специальные символы. Класс `TagBuilder` является частью сборки

`System.Web.WebPages.Mvc`, но так как в нем используется функция *миграции типов* (*type forwarding*), кажется, что он является частью сборки `System.Web.Mvc`. Visual Studio добавляет в проекты MVC обе сборки, и использовать класс `TagBuilder` достаточно легко, хотя его и нет в документации Microsoft Developer Network (MSDN) API.

Чтобы создать новый экземпляр `TagBuilder`, передайте в конструктор как параметр имя элемента HTML, который хотите создать. В классе `TagBuilder` не нужно использовать угловые скобки (< и >), то есть создать элемент `ul` можно следующим образом:

```
TagBuilder tag = new TagBuilder("ul");
```

Наиболее часто используемые члены класса `TagBuilder` описаны в таблице 19-3.

**Таблица 19-3:** Члены класса `TagBuilder`

Член	Описание
<code>Innerhtml</code>	Свойство, которое позволяет записать содержимое элемента как строку HTML. Значение, присвоенное этому свойству, не будет закодировано, что означает, что с его помощью можно создавать вложенные элементы HTML.
<code>SetInnerText(string)</code>	Устанавливает текстовое содержимое элемента HTML. Параметр <code>string</code> кодируется для безопасности отображения (см. раздел о кодировании строк HTML ранее в этой главе).
<code>AddCssClass(string)</code>	Добавляет класс CSS к элементу HTML.
<code>MergeAttribute(string, string, bool)</code>	Добавляет атрибут к элементу HTML. Первый параметр - это имя атрибута, второй - его значение. Параметр <code>bool</code> определяет, нужно ли заменить существующий атрибут с таким же названием.

Результатом вспомогательного метода HTML является объект `MvcHtmlString`, содержимое которого записывается непосредственно в ответ клиенту. В нашем примере мы передаем результат метода `TagBuilder.ToString` в конструктор объекта `MvcHtmlString`, как показано здесь:

```
return new MvcHtmlString(tag.ToString());
```

Этот оператор генерирует фрагмент кода HTML, который содержит элементы `ul` и `li`, и возвращает их в движок представлений, чтобы далее они были записаны в ответ.

### Используем пользовательский внешний вспомогательный метод

Применение пользовательских внешних вспомогательных методов немного отличается от внутренних. В листинге 19-6 показаны изменения, которые мы внесли в файл `/Views/Home/Index.cshtml` для замены внутреннего вспомогательного метода на внешний.

**Листинг 19-6:** Используем пользовательский внешний вспомогательный метод в файле Index.cshtml

```
@model string
@using HelperMethods.Infrastructure

@{
    Layout = null;
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
        Here are the fruits: @Html.ListArrayItems((string[])ViewBag.Fruits)
    </div>
    <div>
        Here are the cities: @Html.ListArrayItems((string[])ViewBag.Cities)
    </div>
    <div>
        Here is the message:
        <p>@Model</p>
    </div>
</body>
</html>
```

Убедитесь, что в файле есть пространство имен, содержащее вспомогательный метод. Чтобы добавить его, мы использовали тег `@using`, но если вы разрабатываете много пользовательских вспомогательных методов, то можно добавить данное пространство имен в файл `/Views/Web.config`, чтобы они были доступны во всех представлениях.

Мы ссылаемся на вспомогательный метод с помощью `@Html.<helper>`, где `<helper>` - это имя метода расширения, в данном случае `Html.ListArrayItems`. Часть `Html` в этом выражении ссылается на свойство, которое определено в базовом классе представления и возвращает объект `HtmlHelper`, то есть тип, к которому мы применили метод расширения в листинге 19-5.

Мы передаем данные во внешний вспомогательный метод таким же образом, как и во внутренний или любой другой метод C#. Нужно только привести динамических свойства объекта `ViewBag` к типу, определенному этим внешним методом - в данном случае, к массиву строк. Пусть это и не такой красивый синтаксис, как во внутренних вспомогательных методах, но только так можно создать удобный многоразовый вспомогательный метод.

## Когда использовать вспомогательные методы

Теперь, когда вы уже видели вспомогательные методы в действии, вы можете задать вопрос: когда лучше использовать их в предпочтение частичным представлениям или дочерним действиям, тем более, что все эти средства дублируют функциональность друг друга.

Мы используем вспомогательные методы только для того, чтобы уменьшить количество дублированного кода в представлениях, и только для самого простого кода, как в приведенном примере. Для более сложной разметки и кода мы используем частичные представления, а если необходимо выполнить какие-либо манипуляции с моделью данных - дочерние действия. Мы советуем вам придерживаться такой же схемы и использовать вспомогательные методы только для самых простых решений (в тех случаях, когда количество операторов C# будет большим или даже превысит количество элементов HTML, мы будем переключаться на дочерние действия).

## Управляем кодировкой строк в вспомогательных методах

В MVC Framework для защиты от вредоносного ввода применяется автоматическое кодирование, которое позволяет гарантировать безопасность добавления данных на страницу. Пример кодирования вы можете увидеть в листинге 19-7, где мы передаем потенциально небезопасную строку в представление в качестве модели объекта (в листинге повторяется код контроллера Home).

### Листинг 19-7: Контроллер Home

```
using System.Web.Mvc;

namespace HelperMethods.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewBag.Fruits = new string[] { "Apple", "Orange", "Pear" };
            ViewBag.Cities = new string[] { "New York", "London", "Paris" };

            string message = "This is an HTML element: <input>";

            return View((object) message);
        }
    }
}
```

Объект модели содержит допустимый элемент HTML, но после обработки Razor мы видим следующий код:

```
<div>
    Here is the message:
    <p>This is an HTML element: &lt;input&gt;</p>
</div>
```

Это основная мера безопасности, которая запрещает браузеру интерпретировать значения данных как действительную (допустимую) разметку, так как это является основой для распространенной формы атак, при которой злоумышленники пытаются изменить поведение приложения, пытаясь добавлять свой собственный код HTML или разметку JavaScript. Razor автоматически кодирует значения данных, когда они используются в представлении, но так как вспомогательные методы должны генерировать HTML, то они пользуются более высоким уровнем доверия со стороны движка представления - и в силу этого требуют более пристального внимания.

### Описание проблемы

Чтобы дать вам представление о проблеме, связанной с кодировкой строк, мы создали новый вспомогательный метод в классе `CustomerHelpers`, как показано в листинге 19-8. Этот вспомогательный метод принимает в качестве параметра строку и генерирует такой же HTML, который мы включили в представление `Index`.

### Листинг 19-8: Определяем новый вспомогательный метод

```
using System.Web.Mvc;
using System;

namespace HelperMethods.Infrastructure
{
    public static class CustomHelpers
    {
```

```

public static MvcHtmlString ListArrayItems(this HtmlHelper html, string[] list)
{
    TagBuilder tag = new TagBuilder("ul");
    foreach (string str in list)
    {
        TagBuilder itemTag = new TagBuilder("li");
        itemTag.SetInnerText(str);
        tag.InnerHtml += itemTag.ToString();
    }
    return new MvcHtmlString(tag.ToString());
}

public static MvcHtmlString DisplayMessage(this HtmlHelper html, string msg)
{
    string result = String.Format("This is the message: <p>{0}</p>", msg);
    return new MvcHtmlString(result);
}
}
}
}

```

Мы используем метод `String.Format`, чтобы создать разметку HTML и передать результат в качестве аргумента в конструктор `MvcHtmlString`. В листинге 19-9 показаны изменения в представлении `/View/Home/Index.cshtml`, с помощью которых мы будем использовать новый вспомогательный метод (а также выделим контент, который генерируется вспомогательным методом).

**Листинг 19-9:** Используем вспомогательный метод `DisplayMessage` в представлении `Index`

```

@model string
@using HelperMethods.Infrastructure

@{
    Layout = null;
}

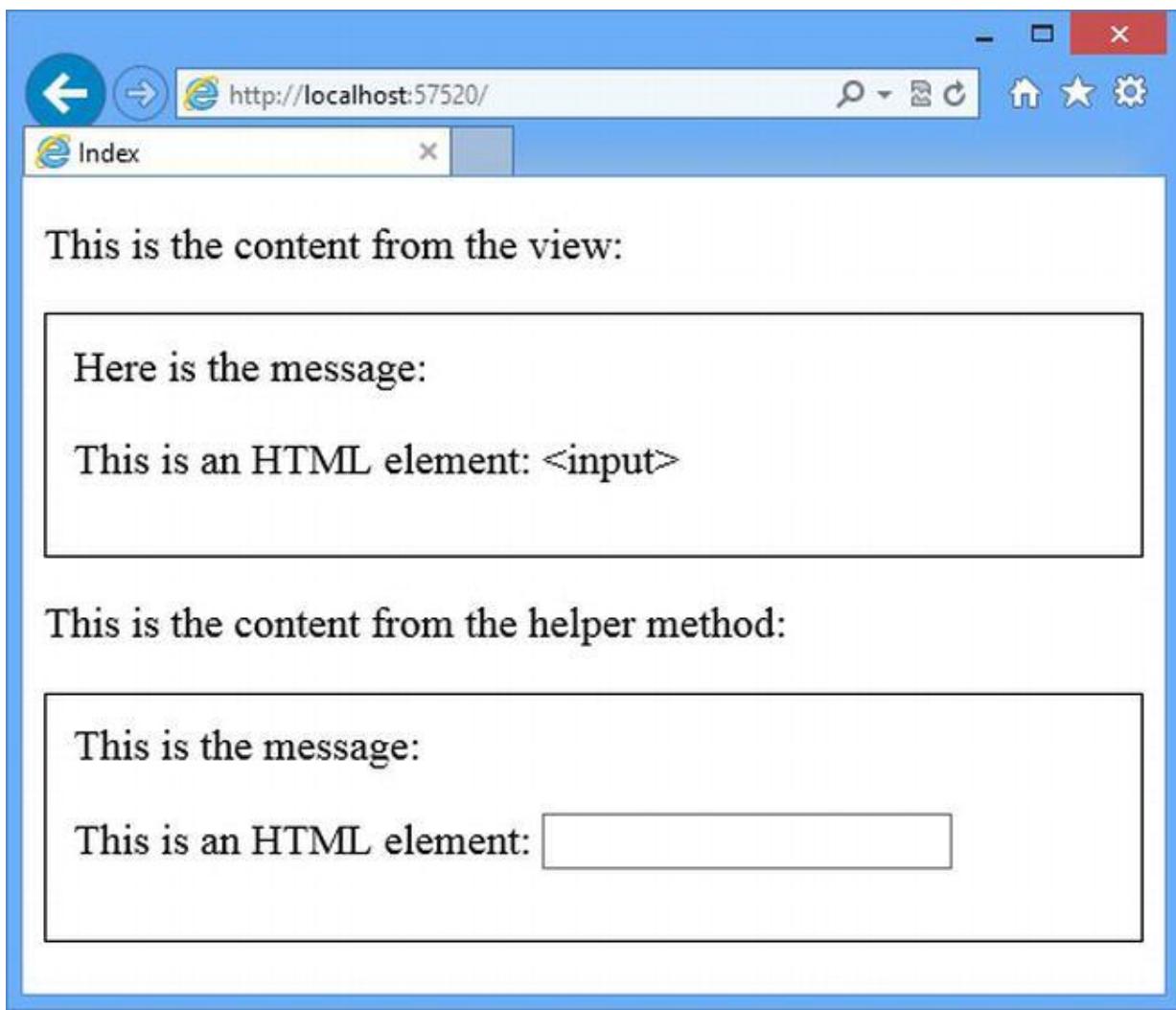
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <p>This is the content from the view:</p>
    <div style="border: thin solid black; padding: 10px">
        Here is the message:
        <p>@Model</p>
    </div>

    <p>This is the content from the helper method:</p>
    <div style="border: thin solid black; padding: 10px">
        @Html.DisplayMessage(Model)
    </div>
</body>
</html>

```

Вы можете увидеть результат применения вспомогательного метода, запустив приложение, как показано на рисунке 19-3.

**Рисунок 19-3:** Сравниваем кодировку значений данных



Движок представлений считает безопасным контент, который генерируется вспомогательным методом, что приводит к нежелательным последствиям: браузер отображает элемент `input`, который можно использовать для взлома приложения.

### Кодируем контент, генерируемый вспомогательными методами

Есть несколько способов решения этой проблемы, и выбор между ними зависит от характера контента, который генерируется вспомогательным методом.

Самое простое решение - изменить тип вывода вспомогательного метода на `string`, как показано в листинге 19-10. Это сообщает движку представления, что контент не является безопасным и должен быть закодирован, прежде чем он будет добавлен в представление.

**Листинг 19-10:** Razor кодирует контент, созданный вспомогательным методом

```
using System.Web.Mvc;
using System;

namespace HelperMethods.Infrastructure
{
    public static class CustomHelpers
    {
        public static MvcHtmlString ListArrayItems(this HtmlHelper html, string[] list)
        {
```

```

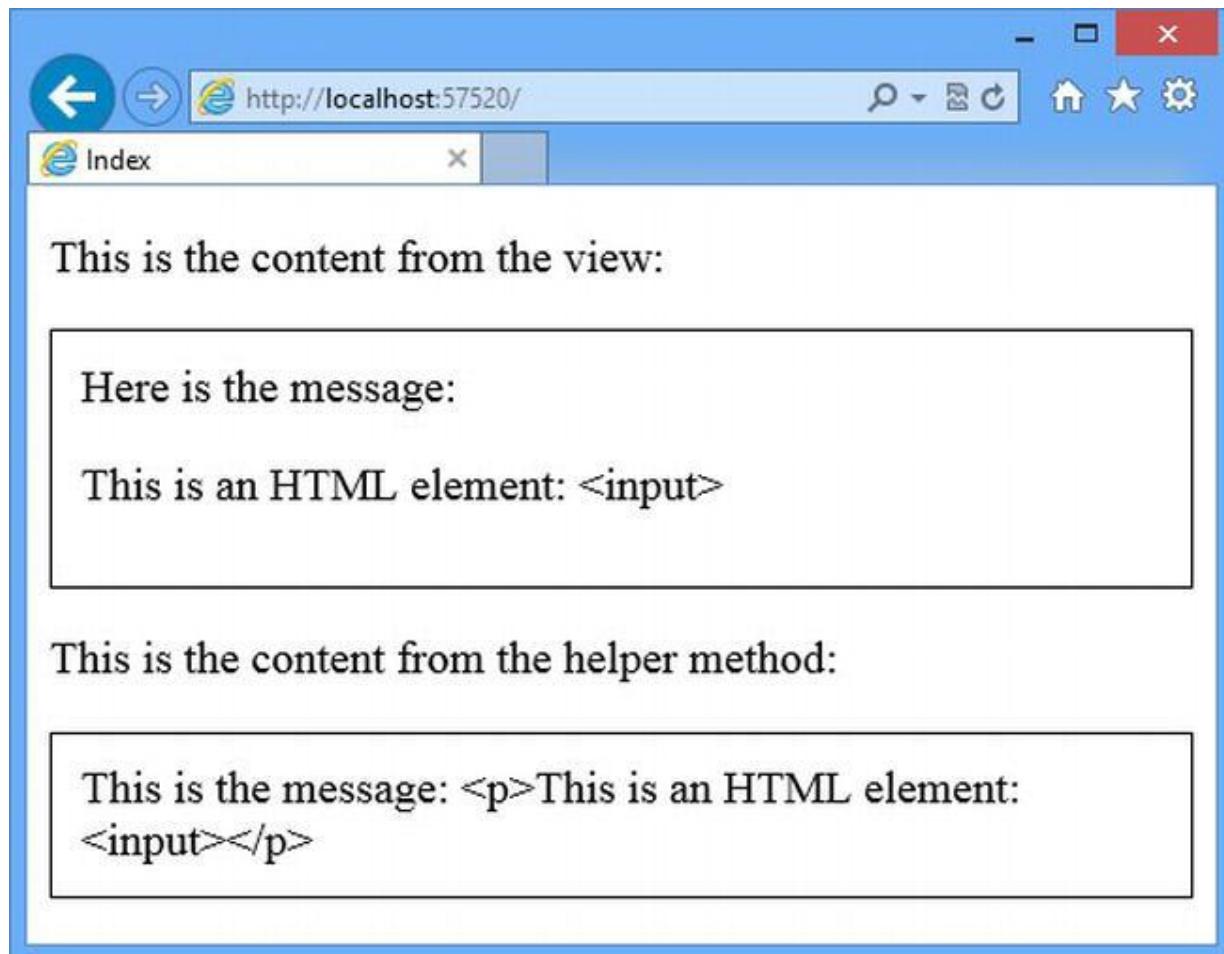
TagBuilder tag = new TagBuilder("ul");
foreach (string str in list)
{
    TagBuilder itemTag = new TagBuilder("li");
    itemTag.SetInnerText(str);
    tag.InnerHtml += itemTag.ToString();
}
return new MvcHtmlString(tag.ToString());
}

public static string DisplayMessage(this HtmlHelper html, string msg)
{
    return String.Format("This is the message: <p>{0}</p>", msg);
}
}
}

```

Таким образом, Razor будет кодировать весь контент, который возвращается вспомогательным методом. В целом это очень удобно, за исключением тех случаев, когда необходимо создать элементы HTML, как в данном примере. Результат показан на рисунке 19-4.

**Рисунок 19-4:** Движок представлений кодирует ответ вспомогательного метода



Мы решили проблему с элементом `input`, но наши элементы `p` также были закодированы, и мы получили не совсем то, что хотели. В таких ситуациях нужно быть более избирательными и кодировать только значения данных. Это показано в листинге 19-11.

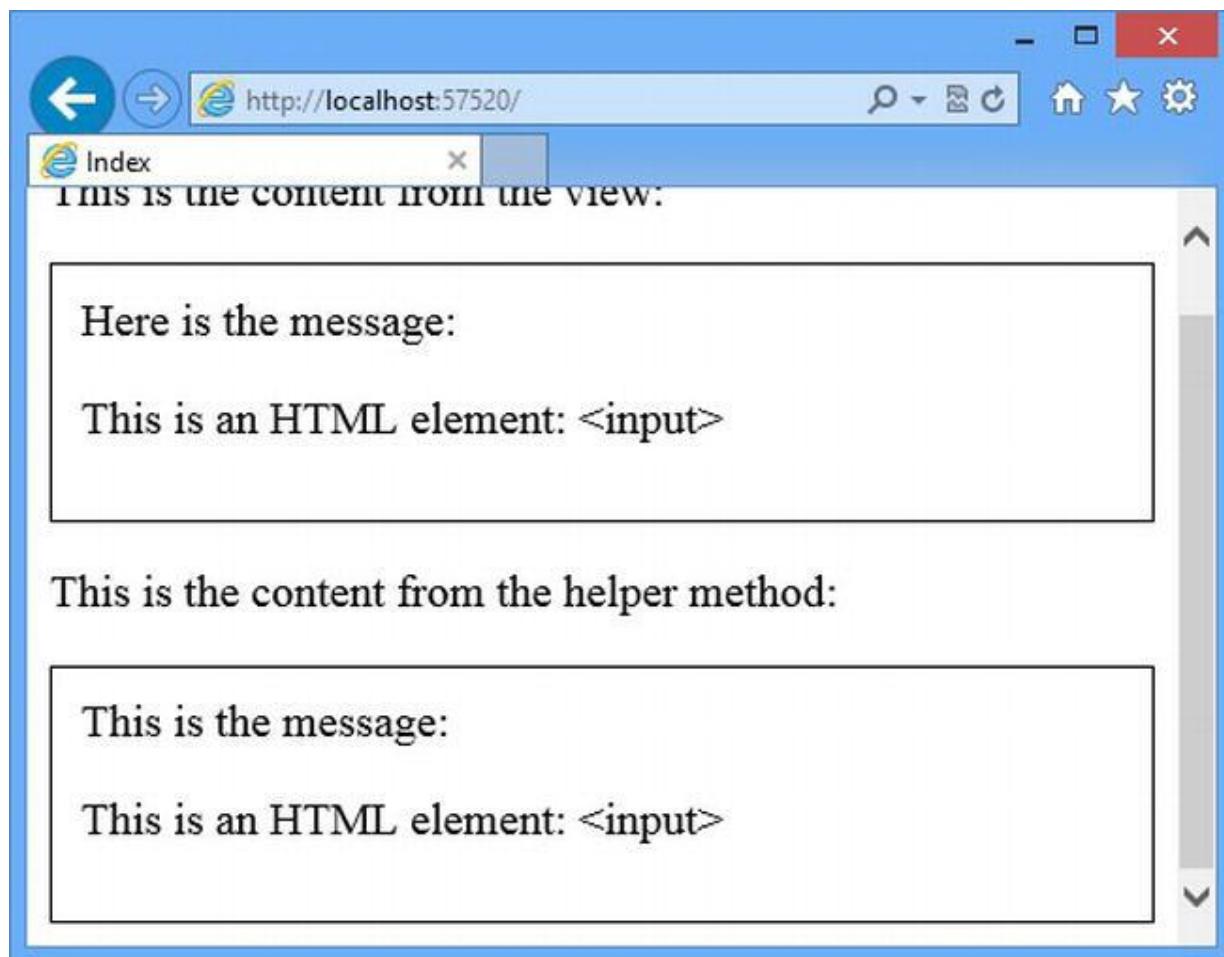
**Листинг 19-11:** Применяем выборочное кодирование данных во вспомогательном методе

```
public static MvcHtmlString DisplayMessage(this HtmlHelper html, string msg)
{
    string encodedMessage = html.Encode(msg);
    string result = String.Format(
        "This is the message: <p>{0}</p>", encodedMessage);
    return new MvcHtmlString(result);
}
```

Класс `HtmlHelper` создает экземпляр метода под названием `Encode`, который решает поставленную задачу и кодирует строковое значение, так что оно может быть безопасно использоваться в представлении. Однако, не забывайте его использовать - в качестве напоминания мы явно кодируем все значения данных в начале метода, предполагая, что вы будете следовать данному подходу.

Результат этого изменения показан на рисунке 19-5, на котором вы можете увидеть, что контент, генерируемый внешним вспомогательным методом, совпадает с контентом, который генерируется с помощью значения модели непосредственно в представлении.

**Рисунок 19-5:** Эффект выборочного кодирования контента во внешнем вспомогательном методе



# Использование встроенных вспомогательных методов

MVC Framework включает в себя несколько встроенных вспомогательных методов, предназначенных для создания форм HTML. В следующих разделах мы рассмотрим эти вспомогательные методы в действии и научимся их использовать.

## Создаем элементы form

Одним из наиболее распространенных способов взаимодействия с пользователем в веб-приложении являются HTML-формы, которые можно создавать с помощью целого ряда вспомогательных методов. Чтобы продемонстрировать методы, предназначенные для работы с формами, мы внесли некоторые дополнения в наш пример проекта. Для начала мы создали новый класс Person.cs в папке Models. Содержимое этого файла показано в листинге 19-12 – тип Person будет нашей моделью представления, с помощью которой мы продемонстрируем связанные с формами вспомогательные методы, а типы Address и Role помогут нам продемонстрировать некоторые более сложные функции.

**Листинг 19-12:** Модель Person

```
using System;

namespace HelperMethods.Models
{
    public class Person
    {
        public int PersonId { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime BirthDate { get; set; }
        public Address HomeAddress { get; set; }
        public bool IsApproved { get; set; }
        public Role Role { get; set; }
    }

    public class Address
    {
        public string Line1 { get; set; }
        public string Line2 { get; set; }
        public string City { get; set; }
        public string PostalCode { get; set; }
        public string Country { get; set; }
    }

    public enum Role
    {
        Admin,
        User,
        Guest
    }
}
```

Мы также добавили новые методы действия в контроллер Home, в которых будут использоваться объекты модели и которые показаны в листинге 19-13.

### Листинг 19-13: Добавляем методы действий в контроллер Home

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

namespace HelperMethods.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewBag.Fruits = new string[] {"Apple", "Orange", "Pear"};
            ViewBag.Cities = new string[] {"New York", "London", "Paris"};
            string message = "This is an HTML element: <input>";
            return View((object) message);
        }

        public ActionResult CreatePerson()
        {
            return View(new Person());
        }

        [HttpPost]
        public ActionResult CreatePerson(Person person)
        {
            return View(person);
        }
    }
}
```

Это стандартный подход к работе с HTML-формами, который включает два метода. Здесь мы полагаемся на механизм связывания данных, предполагая, что MVC Framework создаст объект Person из данных формы и передаст его в метод действия с помощью атрибута `HttpPost`. (Атрибут `HttpPost` рассматривается в главе 16, а связывание данных - в главе 22).

Мы никак не обрабатываем данные формы, потому что хотим сосредоточиться на том, как генерировать элементы в представлении. Наш метод действия `HttpPost` просто вызывает метод `View` и передает в него объект `Person`, который он получил в качестве параметра; таким образом мы отображаем пользователю введенные им в форму данные.

Для начала мы создадим стандартную HTML-форму вручную, а затем покажем вам, как заменить различные ее части с помощью вспомогательных методов. Первоначальную версию формы вы можете увидеть в листинге 19-14; в нем показано содержимое представления `CreatePerson.cshtml`, которое мы создали в папке `/Views/Home`.

### Листинг 19-14: Первоначальная версия формы HTML

```
@model HelperMethods.Models.Person

@{
    ViewBag.Title = "CreatePerson";
}



## CreatePerson



<form action="/Home/CreatePerson" method="post">
    <div class="dataElem">
        <label>PersonId</label>
        <input name="personId" value="@Model.PersonId"/>
    </div>
    <div class="dataElem">
```

```

<label>First Name</label>
<input name="FirstName" value="@Model.FirstName"/>
</div>
<div class="dataElem">
    <label>Last Name</label>
    <input name="lastName" value="@Model.LastName"/>
</div>
<input type="submit" value="Submit" />
</form>

```

Данное представление содержит стандартную форму, созданную вручную, в которой мы установили значение атрибута `value` элементов `input`, используя объект модели.

#### Подсказка

*Обратите внимание, что мы установили атрибут `name` во всех элементах `input` так, чтобы он соответствовал свойству модели, которое отображается элементом `input`. Атрибут `input` используется стандартным механизмом связывания MVC Framework для того, чтобы выяснить, какие элементы `input` содержат значения для свойств типа модели во время обработки запроса `post` и, если вы не укажете атрибут `name`, форма не будет работать должным образом. В главе 22 мы подробно описываем механизм связывания данных, а также способы изменения его поведения.*

---

Мы отредактировали содержимое файла `/Views/Shared/_Layout.cshtml`, как показано в листинге 19-15, чтобы максимально упростить данный пример. Мы удалили теги Razor `@Scripts` и `@Styles`, которые будут описаны в главе 24.

#### Листинг 19-15: Упрощаем макет

```

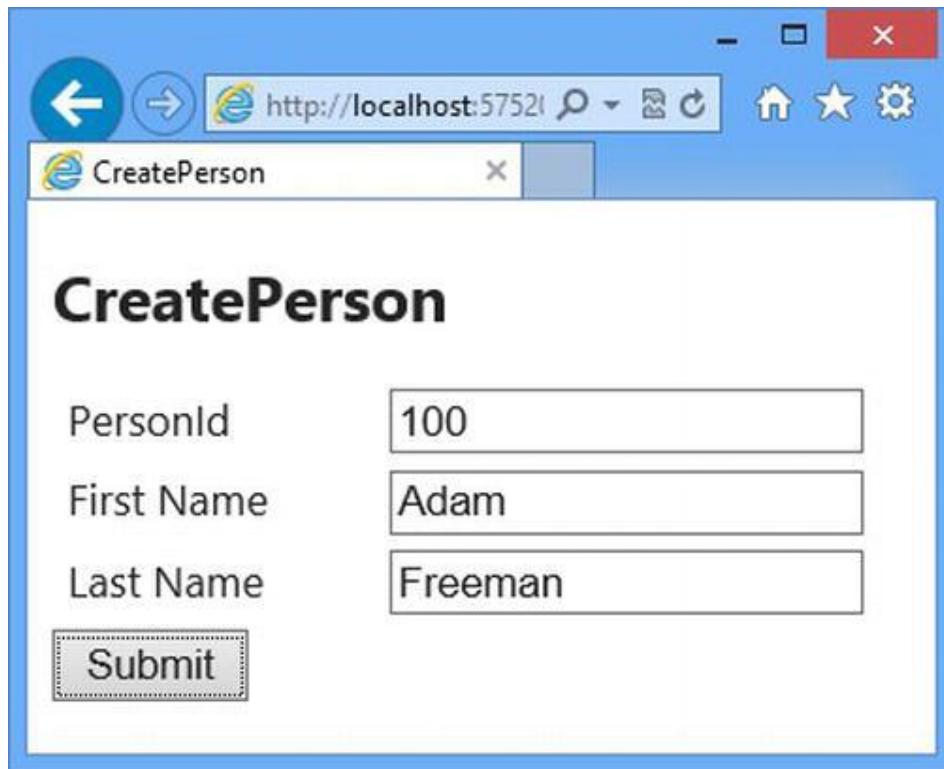
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    <link href="~/Content/Site.css" rel="stylesheet" />
    <style type="text/css">
        label {
            display: inline-block;
            width: 100px;
        }

        .dataElem {
            margin: 5px;
        }
    </style>
</head>
<body>
    @RenderBody()
</body>
</html>

```

Чтобы увидеть базовую функциональность формы, запустите приложение и перейдите по ссылке `/Home/CreatePerson`. HTML-форма с примерами данных показана на рисунке 19-6. Так как данные формы никак не используются в приложении, то после нажатия на кнопку `Submit` они будут просто отображаться пользователю.

**Рисунок 19-6:** Используем простую форму HTML в приложении



В листинге 19-16 показан код HTML, который приложение направило в браузер; мы будем его использовать, чтобы продемонстрировать изменения, вызванные применением вспомогательных методов.

**Листинг 19-16:** Код HTML, отправленный в браузер

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>CreatePerson</title>
    <link href="/Content/Site.css" rel="stylesheet" />
    <style type="text/css">
        label {
            display: inline-block;
            width: 100px;
        }

        .dataElem {
            margin: 5px;
        }
    </style>
</head>
<body>
    <h2>CreatePerson</h2>
    <form action="/Home/CreatePerson" method="post">
        <div class="dataElem">
            <label>PersonId</label>
            <input name="personId" value="0" />
        </div>
        <div class="dataElem">
            <label>First Name</label>
```

```

<input name="FirstName" />
</div>
<div class="dataElem">
    <label>Last Name</label>
    <input name="lastName" />
</div>
<input type="submit" value="Submit" />
</form>
</body>
</html>

```

#### Примечание

*Используя вспомогательные методы, не обязательно генерировать такие HTML-элементы, как `form` и `input`. Если хотите, вы можете закодировать их, используя статические HTML-теги, и заполнить значения с помощью объектов `ViewData` или `ViewModel`, как это сделали мы в данном разделе. В следующих разделах мы покажем, как с помощью вспомогательных методов создать чистый HTML-код без каких-либо специальных значений атрибутов. Это не означает, что вы обязаны их использовать, но с их помощью очень легко гарантировать, что HTML будет находиться в синхронизации с приложением, так что, например, изменения в конфигурации будут автоматически отражаться в формах. Вспомогательные методы нужно использовать для удобства, а не потому, что они создают особенный или специальный HTML-код, и вас никто не принуждает их использовать, если они не соответствуют вашему стилю разработки.*

## Создаем элементы form

Два из наиболее полезных (и наиболее часто используемых) вспомогательных метода - это `Html.BeginForm` и `Html.EndForm`. Они создают теги формы HTML и генерируют для нее допустимый атрибут `action`, основываясь на механизме маршрутизации приложения.

Существует 13 различных версий метода `BeginForm`, что позволяет нам очень точно указать, как должен быть сгенерирован элемент формы. Для нашего приложения потребуется самая базовая версия, которая не принимает аргументов. Она создает элемент формы, атрибут `action` которого гарантирует, что форма будет отправлена к тому же методу действия, который визуализировал текущее представление. В листинге 19-17 показано, как мы применили эту перегруженную версию `BeginForm`, а также вспомогательный метод `EndForm` (для `EndForm` есть только одно определение, и он просто закрывает элемент формы, добавляя в представление `</form>`).

#### Листинг 19-17: Используем вспомогательные методы `BeginForm` и `EndForm`

```

@model HelperMethods.Models.Person
 @{
    ViewBag.Title = "CreatePerson";
}
<h2>CreatePerson</h2>
@Html.BeginForm()
<div class="dataElem">
    <label>PersonId</label>
    <input name="personId" value="@Model.PersonId"/>
</div>
<div class="dataElem">
    <label>First Name</label>
    <input name="FirstName" value="@Model.FirstName"/>
</div>

```

```

<div class="dataElem">
    <label>Last Name</label>
    <input name="lastName" value="@Model.LastName"/>
</div>
<input type="submit" value="Submit" />
@{ Html.EndForm(); }

```

Обратите внимание, что мы должны были обработать вызов к вспомогательному методу `EndForm` как оператор C#, так как метод `EndForm` записывает свой тег непосредственно в вывод, а не возвращает результат, который можно добавить в представление, как это делает `BeginForm`.

С точки зрения дизайна решение просто ужасное, но это не имеет значения, потому что метод `EndForm` используется редко. Более распространенный подход показан в листинге 19-18, где вызов к вспомогательному методу `BeginForm` заключается в выражение `using`. В конце блока `using` среда выполнения .NET вызывает для объекта, который возвращается методом `BeginForm`, метод `Dispose`, который и вызывает метод `EndForm`. (Чтобы увидеть, как это работает, загрузите исходный код MVC Framework и рассмотрите класс `System.Web.Mvc.Html.FormExtensions`).

#### Листинг 19-18: Создаем самозакрывающуюся форму

```

@model HelperMethods.Models.Person
 @{
    ViewBag.Title = "CreatePerson";
}
<h2>CreatePerson</h2>
@using (Html.BeginForm())
{
    <div class="dataElem">
        <label>PersonId</label>
        <input name="personId" value="@Model.PersonId"/>
    </div>
    <div class="dataElem">
        <label>First Name</label>
        <input name="FirstName" value="@Model.FirstName"/>
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        <input name="lastName" value="@Model.LastName"/>
    </div>
    <input type="submit" value="Submit" />
}

```

Этот подход, известный как самозакрывающаяся форма (*self-closing form*), мы чаще всего используем в своих проектах – нам нравится то, что форма содержится в блоке кода и становится ясно, какие элементы будут находиться между открывающим и закрывающим тегами формы.

Остальные 12 версий метода `BeginForm` позволяют изменять различные аспекты элемента формы. В этих перегруженных версиях есть много повторов, так как они указывают различные детали, касающиеся создания формы. В таблице 19-4 мы перечислили наиболее важные версии, которые вы будете использовать в приложениях регулярно. Другие перегруженные версии метода `BeginForm` нужны для совместимости с версиями MVC Framework, которые были выпущены до того, как в C# появилась поддержка создания динамических объектов.

**Таблица 19-4:** Перегруженные версии вспомогательного метода BeginForm

Перегруженная версия	Описание
BeginForm()	Создает форму, которая отправляет данные в метод действия, который ее сгенерировал.
BeginForm( <b>action</b> , <b>controller</b> )	Создает форму, которая отправляет данные в метод действия и контроллер, указанные в параметрах.
BeginForm( <b>action</b> , <b>controller</b> , <b>method</b> )	Как и предыдущие перегруженные версии, но позволяет указать значение атрибута <b>method</b> , используя значение из перечисления System.Web.Mvc.FormMethod.
BeginForm( <b>action</b> , <b>controller</b> , <b>method</b> , <b>attributes</b> )	Как и предыдущие перегруженные версии, но позволяет задавать атрибуты элемента формы с помощью объекта, свойства которого используются в качестве имен атрибутов.
BeginForm( <b>action</b> , <b>controller</b> , <b>routeValues</b> , <b>method</b> , <b>attributes</b> )	Как и предыдущие перегруженные версии, но позволяет вам указать значения для сегментов переменных маршрутов в конфигурации маршрутизации в приложении с помощью объекта, свойства которого соответствуют переменным маршрутизации.

Мы показали вам простейший вариант метода BeginForm, который полностью подходит для нашего приложения. В листинге 19-19 показана самая сложная версия, в которой мы указали дополнительную информацию для создания элемента формы.

**Листинг 19-19:** Используем самую сложную перегруженную версию метода BeginForm

```
@model HelperMethods.Models.Person
 @{
    ViewBag.Title = "CreatePerson";
}
<h2>CreatePerson</h2>
@using (Html.BeginForm("CreatePerson", "Home",
    new { id = "MyIdValue" }, FormMethod.Post,
    new { @class = "personClass", data_formType = "person" }))
{
    <div class="dataElem">
        <label>PersonId</label>
        <input name="personId" value="@Model.PersonId"/>
    </div>
    <div class="dataElem">
        <label>First Name</label>
        <input name="FirstName" value="@Model.FirstName"/>
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        <input name="lastName" value="@Model.LastName"/>
    </div>
    <input type="submit" value="Submit" />
}
```

В этом примере явно указаны некоторые детали, такие как имя действия и контроллера, которые в противном случае были бы выведены MVC Framework автоматически. Мы также уточнили, что

форма должна быть отправлена с помощью метода HTTP POST, который использовался бы в любом случае.

Самые интересные аргументы - те, которые устанавливают значения для переменных маршрута и задают атрибуты для элемента form. С помощью аргумента значения маршрута мы указали значение переменной сегмента id в стандартном маршруте, который был добавлен Visual Studio в файл /App\_Start/RouteConfig.cs при создании проекта. Мы также определили атрибуты class и data (атрибуты data - это пользовательские атрибуты, которых можно добавлять к элементам, чтобы они обрабатывали контент HTML). Вот тег формы HTML, который генерирует этот вызов к BeginForm:

```
<form action="/Home/CreatePerson/MyIdValue" class="personClass" data-formType="person"
method="post">
```

Как видите, значение атрибута id было добавлено к целевому URL, а атрибуты class и data - к элементу формы. Обратите внимание, что в нашем вызове к BeginForm мы указали атрибут data\_formType, но в итоге получили атрибут data-formType. Нельзя указывать в динамическом объекте имена свойств с дефисами, поэтому мы используем символ подчеркивания, который на выходе автоматически преобразуется в дефис, аккуратно обходя несоответствие между синтаксисом C# и HTML. (И, конечно, мы должны поставить префикс @ перед именем свойства class, чтобы использовать зарезервированное ключевое слово C# в качестве имени свойства для атрибута класса).

## Определяем маршрут, который будет использоваться формой

Когда вы используете метод BeginForm, MVC Framework будет искать в конфигурации маршрутизации первый маршрут, с помощью которого она сможет сгенерировать ссылку на запрашиваемое действие или контроллер – в общем, выбор маршрута платформа полностью берет на себя. Если вы хотите указать конкретный маршрут, можете использовать вспомогательный метод BeginRouteForm. Чтобы продемонстрировать его работу, мы добавили новый маршрут в файл /App\_Start/RouteConfig.cs, как показано в листинге 19-20.

**Листинг 19-20:** Добавляем новый маршрут в файл RouteConfig.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace HelperMethods
{
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
            routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new
                {
                    controller = "Home",
                    action = "Index",
                    id = UrlParameter.Optional
                }
            );
            routes.MapRoute(
                name: "FormRoute",
```

```

        url: "app/forms/{controller}/{action}"
    );
}
}
}

```

Если мы вызовем метод `BeginForm` при такой конфигурации маршрутизации, то получим элемент `form` с атрибутом `action`, который содержит URL, созданный по стандартному маршруту. В листинге 19-21 с помощью метода `BeginRouteForm` мы указали, что нужно использовать новый маршрут.

#### **Листинг 19-21:** Указываем маршрут для создания URL

```

@model HelperMethods.Models.Person
{
    ViewBag.Title = "CreatePerson";
}
<h2>CreatePerson</h2>
@using (Html.BeginRouteForm("FormRoute", new { }, FormMethod.Post,
    new { @class = "personClass", data_formType = "person" }))
{
    <div class="dataElem">
        <label>PersonId</label>
        <input name="personId" value="@Model.PersonId"/>
    </div>
    <div class="dataElem">
        <label>First Name</label>
        <input name="FirstName" value="@Model.FirstName"/>
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        <input name="lastName" value="@Model.LastName"/>
    </div>
    <input type="submit" value="Submit" />
}

```

Теперь мы получаем тег `form`, атрибут `action` которого содержит URL, соответствующий нашему новому маршруту:

```
<form action="/app/forms/Home/CreatePerson" class="personClass"
    data-formType="person" method="post">
```

*Подсказка*

*Для метода `BeginRouteForm` есть целый ряд перегруженных версий, которые позволяют указывать детали элемента `form`, как и для метода `BeginForm`. Они повторяют структуру версий `BeginForm`. Более подробную информацию можно узнать из документации API.*

## **Создаем элементы input**

Элемент HTML `form` совершенно бесполезен без элементов `input`. В таблице 19-5 показаны базовые вспомогательные методы для создания элементов ввода, а также приводятся примеры генерируемого ими кода HTML. Для всех этих вспомогательных методов первый аргумент присваивает значения атрибутам `id` и `name` элемента `input`, а второй присваивает значение атрибуту `value`.

**Таблица 19-5:** Базовые вспомогательные методы для создания элементов ввода

Элемент HTML	Пример
Чекбокс	Html.CheckBox("myCheckbox", false) <b>Вывод:</b> <input id="myCheckbox" name="myCheckbox" type="checkbox" value="true" /><input name="myCheckbox" type="hidden" value="false" />
Скрытое поле	Html.Hidden("myHidden", "val") <b>Вывод:</b> <input id="myHidden" name="myHidden" type="hidden" value="val" />
Переключатель	Html.RadioButton("myRadioButton", "val", true) <b>Вывод:</b> <input checked="checked" id="myRadioButton" name="myRadioButton" type="radio" value="val" />
Пароль	Html.Password("myPassword", "val") <b>Вывод:</b> <input id="myPassword" name="myPassword" type="password" value="val" />
Текстовая область (многострочное текстовое поле)	Html.TextArea("myTextarea", "val", 5, 20, null) <b>Вывод:</b> <textarea cols="20" id="myTextarea" name="myTextarea" rows="5"> val</textarea>
Текстовое поле	Html.TextBox("myTextBox", "val") <b>Вывод:</b> <input id="myTextBox" name="myTextBox" type="text" value="val" />

Все эти вспомогательные методы являются перегруженными версиями. В таблице показан самый простой вариант, но, чтобы создать атрибуты HTML, вы можете указать дополнительные объекты, как мы это делали в предыдущем разделе для элемента `form`.

#### Примечание

*Обратите внимание, что вспомогательный метод для чекбокса (`Html.CheckBox`) визуализирует два элемента `input`. Он визуализирует чекбокс и скрытый элемент ввода с таким же названием. Это необходимо потому, что браузеры не отправляют значения чекбокса, если он не был отмечен. Наличие скрытого элемента управления гарантирует, что в таких случаях MVC Framework получит значение из скрытого поля.*

В листинге 19-22 показано, как мы использовали базовые вспомогательные методы для создания элементов ввода.

**Листинг 19-22:** Используем базовые вспомогательные методы для создания элементов ввода

```
@model HelperMethods.Models.Person
{
    ViewBag.Title = "CreatePerson";
}

<h2>CreatePerson</h2>
```

```

@using (Html.BeginRouteForm("FormRoute", new {}, FormMethod.Post,
    new {@class = "personClass", data_formType = "person"}))
{
    <div class="dataElem">
        <label>PersonId</label>
        @Html.TextBox("personId", @Model.PersonId)
    </div>
    <div class="dataElem">
        <label>First Name</label>
        @Html.TextBox("firstName", @Model.FirstName)
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        @Html.TextBox("lastName", @Model.LastName)
    </div>
    <input type="submit" value="Submit" />
}

```

Элементы HTML `input`, которые генерирует это представление, показаны в листинге 19-23. Вывод очень похож на наш первоначальный элемент `form`, но здесь уже появились некоторые намеки на MVC Framework – например, атрибуты `data`, которые добавляют валидацию формы (валидацию мы рассматривали в главе 2 и вернемся к этой теме в главе 23).

#### **Листинг 19-23:** Элементы `input`, созданные с помощью базовых вспомогательных методов ввода

```

<form action="/app/forms/Home/CreatePerson" class="personClass" data-formtype="person"
method="post">
    <div class="dataElem">
        <label>PersonId</label>
        <input data-val="true" data-val-number="The field PersonId must be a number."
               data-val-required="The PersonId field is required." id="personId"
               name="personId" type="text" value="0" />
    </div>
    <div class="dataElem">
        <label>First Name</label>
        <input id="firstName" name="firstName" type="text" value="" />
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        <input id="lastName" name="lastName" type="text" value="" />
    </div>
    <input type="submit" value="Submit" />
</form>

```

*Подсказка*

*Чтобы создать атрибуты `data`, которые поддерживают валидацию форм для всего приложения, установите параметру `UnobtrusiveJavaScriptEnabled` значение `false` в файле `Web.config`. Отключить эту функцию для отдельных представлений можно вызовом `Html.EnableClientValidation(false)` в блоке кода `Razor`. Пример этого вы сможете увидеть в главе 23, где мы отключим атрибуты `data`, чтобы создавать элементы HTML с помощью более продвинутых вспомогательных методов.*

---

#### **Создаем элемент `input` из свойства модели**

Вспомогательные методы, которые мы использовали в предыдущем разделе, неплохо справляются со своими обязанностями, но мы все еще должны гарантировать, что значение первого аргумента соответствует значению модели, которую мы передаем в качестве второго аргумента. В противном

случае MVC Framework не сможет восстановить объект модели из данных формы, поскольку атрибуты `name` и значения `forms` элементов `input` не будут совпадать.

Для каждого из методов, перечисленных в таблице 19-5, существует альтернативная перегруженная версия, которая принимает строковый аргумент `single`, который мы использовали в листинге 19-24.

**Листинг 19-24:** Создаем элемент `input` с помощью имени свойства модели

```
@model HelperMethods.Models.Person
{
    ViewBag.Title = "CreatePerson";
}
<h2>CreatePerson</h2>
@using (Html.BeginRouteForm("FormRoute", new { }, FormMethod.Post,
    new { @class = "personClass", data_formType = "person" }))
{
    <div class="dataElem">
        <label>PersonId</label>
        @Html.TextBox("PersonId")
    </div>
    <div class="dataElem">
        <label>First Name</label>
        @Html.TextBox("firstName")
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        @Html.TextBox("lastName")
    </div>
    <input type="submit" value="Submit" />
}
```

Аргумент `string` используется для поиска в `ViewData`, `ViewBag` и `ViewModel` данных, которые используются для создания элемента `input`. Так, например, если вы вызываете `@Html.TextBox("DataValue")`, MVC Framework попытается найти какие-либо данные, которые соответствуют ключу `DataValue`. Проверяются следующие адреса:

- `ViewBag.DataValue;`
- `@Model.DataValue.`

Первое найденное значение используется для установки значения атрибута `value` генерируемого элемента HTML. (Последний адрес, `@Model.DataValue`, проверяется, только если модель представления содержит свойство или поле под названием `DataValue`.)

Если указать строку вида `DataValue.First.Name`, поиск станет более сложным. MVC Framework будет проверять различные адреса, в который `First` и `Name` будут записаны по-разному, например:

- `ViewBag.DataValue.First.Name`
- `ViewBag.DataValue["First"].Name`
- `ViewBag.DataValue["First.Name"]`
- `ViewBag.DataValue["First"]["Name"]`

Напомним, что будет использовано первое найденное значение, на котором поиск будет остановлен. Очевидно, это решение сделано для оптимизации производительности, но, как правило, `ViewBag` содержит немного значений, так что их проверка не занимает много времени.

## Используем строго типизированные вспомогательные методы ввода

Для каждого из базовых вспомогательных методов ввода, которые мы описали в таблице 19-5, имеются соответствующие строго типизированные версии. Они показаны в таблице 19-6 вместе с примерами кода HTML, который они генерируют.

Эти вспомогательные методы используются только в строго типизированных представлениях. (Для краткости мы опустили из таблицы некоторые вспомогательные методы, которые генерируют атрибуты валидации форм на стороне клиента.)

**Таблица 19-6:** Строго типизированные вспомогательные методы ввода

Элемент HTML Пример

Чекбокс	<code>Html.CheckBoxFor(x =&gt; x.IsApproved)</code> Вывод: <code>&lt;input id="IsApproved" name="IsApproved" type="checkbox" value="true" /&gt;&lt;input name="IsApproved" type="hidden" value="false" /&gt;</code>
Скрытое поле	<code>Html.HiddenFor(x =&gt; x.FirstName)</code> Вывод: <code>&lt;input id="FirstName" name="FirstName" type="hidden" value="" /&gt;</code>
Переключатель	<code>Html.RadioButtonFor(x =&gt; x.IsApproved, "val")</code> Вывод: <code>&lt;input id="IsApproved" name="IsApproved" type="radio" value="val" /&gt;</code>
Пароль	<code>Html.PasswordFor(x =&gt; x.Password)</code> Вывод: <code>&lt;input id="Password" name="Password" type="password" /&gt;</code>
Текстовая область	<code>Html.TextAreaFor(x =&gt; x.Bio, 5, 20, new{})</code> Вывод: <code>&lt;textarea cols="20" id="Bio" name="Bio" rows="5"&gt;Bio value&lt;/textarea&gt;</code>
Текстовое поле	<code>Html.TextBoxFor(x =&gt; x.FirstName)</code> Вывод: <code>&lt;input id="FirstName" name="FirstName" type="text" value="" /&gt;</code>

Строго типизированные вспомогательные методы ввода работают с лямбда-выражениями. Значение, которое передается в выражение, - это объект модели представления, и вы можете выбрать поле или свойство, которое будет использоваться для установки атрибута `value`. В листинге 19-25 показано, как мы использовали этот вид вспомогательных методов в представлении `CreatePerson.cshtml`.

**Листинг 19-25:** Используем типизированные вспомогательные методы ввода

```
@model HelperMethods.Models.Person
@{
    ViewBag.Title = "CreatePerson";
}
<h2>CreatePerson</h2>
@using (Html.BeginRouteForm("FormRoute", new { }, FormMethod.Post,
    new { @class = "personClass", data_formType = "person" }))
{
    <div class="dataElem">
        <label>PersonId</label>
        @Html.TextBoxFor(m => m.PersonId)
    </div>
    <div class="dataElem">
        <label>First Name</label>
        @Html.TextBoxFor(m => m.FirstName)
```

```

</div>
<div class="dataElem">
    <label>Last Name</label>
    @Html.TextBoxFor(m => m.LastName)
</div>
<input type="submit" value="Submit" />
}

```

Сгенерированный ими HTML ничем не отличается, но мы предпочитаем использовать строго типизированные вспомогательные методы потому, что они снижают риск возникновения ошибки из-за опечатки в имени свойства.

## Создаем элементы select

В таблице 19-7 показаны вспомогательные методы, которые используются для создания элементов `select`. С их помощью можно создать элемент, позволяющий выбрать один пункт из раскрывающегося списка, или элемент множественного выбора, который позволяет выбрать несколько пунктов. Как для других элементов форм, для этих вспомогательных методов существуют слабо и строго типизированные версии.

**Таблица 19-7:** Вспомогательные методы HTML, которые визуализируют элементы выбора

Элемент HTML	Пример
Раскрывающийся список	<pre>Html.DropDownList("myList", new SelectList(new [] {"A", "B"}),     "Choose") Вывод: &lt;select id="myList" name="myList"&gt; &lt;option     value=""&gt;Choose&lt;/option&gt; &lt;option&gt;A&lt;/option&gt;      &lt;option&gt;B&lt;/option&gt; &lt;/select&gt;</pre>
Раскрывающийся список	<pre>Html.DropDownListFor(x =&gt; x.Gender, new SelectList(new [] {"M",     "F"})) Вывод: &lt;select id="Gender" name="Gender"&gt; &lt;option&gt;M&lt;/option&gt;     &lt;option&gt;F&lt;/option&gt; &lt;/select&gt;</pre>
Список множественного выбора	<pre>Html.ListBox("myList", new MultiSelectList(new [] {"A", "B"})) Вывод:     &lt;select id="myList" multiple="multiple" name="myList"&gt;         &lt;option&gt;A&lt;/option&gt;          &lt;option&gt;B&lt;/option&gt; &lt;/select&gt;</pre>
Список множественного выбора	<pre>Html.ListBoxFor(x =&gt; x.Vals, new MultiSelectList(new [] {"A", "B"}))     Вывод: &lt;select id="Vals" multiple="multiple" name="Vals"&gt;         &lt;option&gt;A&lt;/option&gt;          &lt;option&gt;B&lt;/option&gt; &lt;/select&gt;</pre>

Вспомогательные методы `select` принимают параметры `SelectList` или `MultiSelectList`. Разница между этими классами заключается в том, что для `MultiSelectList` есть опции конструктора, которые позволяют указать, что при первоначальной загрузке страницы необходимо выбрать более одного пункта.

Оба эти классы работают с последовательностью объектов `IEnumerable`. В таблице 19-7 мы создали внутренние массивы, которые содержат пункты списка, но `SelectList` и `MultiSelectList` будут извлекать значения для пунктов списка из объектов, в том числе из объекта модели. Как видите, мы создали элемент `select` для свойства `Role` модели `Person` в листинге 19 - 26.

**Листинг 19-26:** Создаем элемент select для свойства Person.Role

```
@model HelperMethods.Models.Person
{
    ViewBag.Title = "CreatePerson";
}
<h2>CreatePerson</h2>
@using (Html.BeginRouteForm("FormRoute", new {}, FormMethod.Post,
    new { @class = "personClass", data_formType = "person" }))
{
    <div class="dataElem">
        <label>PersonId</label>
        @Html.TextBoxFor(m => m.PersonId)
    </div>
    <div class="dataElem">
        <label>First Name</label>
        @Html.TextBoxFor(m => m.FirstName)
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        @Html.TextBoxFor(m => m.LastName)
    </div>
    <div class="dataElem">
        <label>Role</label>
        @Html.DropDownListFor(m => m.Role,
            new SelectList(Enum.GetNames(typeof(HelperMethods.Models.Role))))
    </div>
    <input type="submit" value="Submit" />
}
```

Мы определили свойство Role, используя значение из перечисления Role, определенного в том же классе. Поскольку объекты SelectList и MultiSelectList работают на объектах I Enumerable, мы должны применить метод Enum.GetNames, чтобы использовать Role enum в качестве источника для элемента select. В листинге 19-27 показан код HTML, который создает наше обновленное представление, в том числе элемент select.

**Листинг 19-27:** Код HTML, который создает представление CreatePerson

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>CreatePerson</title>
    <link href="/Content/Site.css" rel="stylesheet" />
    <style type="text/css">
        label {
            display: inline-block;
            width: 100px;
        }

        .dataElem {
            margin: 5px;
        }
    </style>
</head>
<body>
    <h2>CreatePerson</h2>
    <form action="/app/forms/Home/CreatePerson" class="personClass" data-
formType="person"
        method="post">
        <div class="dataElem">
            <label>PersonId</label>
```

```

<input data-val="true" data-val-number="The field PersonId must be a number."
data-val-required=
The PersonId field is required."
    id="PersonId" name="PersonId" type="text" value="0" />
</div>
<div class="dataElem">
    <label>First Name</label>
    <input id="FirstName" name="FirstName" type="text" value="" />
</div>
<div class="dataElem">
    <label>Last Name</label>
    <input id="LastName" name="LastName" type="text" value="" />
</div>
<div class="dataElem">
    <label>Role</label>
    <select data-val="true" data-val-required="The Role field is required."
        id="Role" name="Role">
        <option selected="selected">Admin</option>
        <option>User</option>
        <option>Guest</option>
    </select>
</div>
<input type="submit" value="Submit" />
</form>
</body>
</html>

```

## Резюме

В этой главе мы рассмотрели вспомогательные методы, которые можно использовать для создания блоков контента в представлениях. Мы показали вам, как создавать собственные внутренние и внешние вспомогательные методы, а затем рассмотрели встроенные вспомогательные методы, с помощью которых можно создавать элементы HTML `form`, `input` и `select`. В следующей главе мы продолжим рассматривать эту тему и научимся использовать шаблонные вспомогательные методы.

# Шаблонные вспомогательные методы

Вспомогательные методы HTML, которые мы рассмотрели в предыдущей главе (такие как `Html.CheckBoxFor` и `Html.TextBoxFor`), генерируют определенный тип элемента. Это значит, что мы должны заранее решить, какой элемент будет представлять данное свойство модели, и вручную обновлять представления, если тип свойства изменится.

В этой главе мы рассмотрим шаблонные вспомогательные методы, которые позволяют нам только указать свойство, которое мы хотим визуализировать, не уточняя, какой элемент HTML для него требуется - это MVC Framework выяснит самостоятельно. Это более гибкий подход к отображению данных пользователю, хотя он и требует немного больше внимания и осторожности.

## Обзор проекта для примера

В этой главе мы продолжим использовать проект `HelperMethod`, который создали в главе 19. Здесь у нас есть класс модели `Person` и пара сопровождающих типов. Чтобы легче было вспомнить, мы перечислили их в листинге 20-1.

**Листинг 20-1:** Типы Person, Address и Role

```
using System;

namespace HelperMethods.Models
{
    public class Person
    {
        public int PersonId { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime BirthDate { get; set; }
        public Address HomeAddress { get; set; }
        public bool IsApproved { get; set; }
        public Role Role { get; set; }
    }

    public class Address
    {
        public string Line1 { get; set; }
        public string Line2 { get; set; }
        public string City { get; set; }
        public string PostalCode { get; set; }
        public string Country { get; set; }
    }

    public enum Role
    {
        Admin,
        User,
        Guest
    }
}
```

Пример проекта содержит очень простой контроллер `Home`, с помощью которого мы отображаем формы и получаем данные от них. Определение класса `HomeController` показано в листинге 20-2.

## Листинг 20-2: Класс HomeController

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

namespace HelperMethods.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewBag.Fruits = new string[] {"Apple", "Orange", "Pear"};
            ViewBag.Cities = new string[] {"New York", "London", "Paris"};
            string message = "This is an HTML element: <input>";
            return View((object) message);
        }

        public ActionResult CreatePerson()
        {
            return View(new Person());
        }

        [HttpPost]
        public ActionResult CreatePerson(Person person)
        {
            return View(person);
        }
    }
}
```

В этой главе мы будем использовать два метода действий CreatePerson, каждый из которых визуализирует представление /Views/Home/CreatePerson.cshtml. В листинге 20-3 показано представление CreatePerson, которое мы создали в конце главы 19.

## Листинг 20-3: Представление CreatePerson

```
@model HelperMethods.Models.Person
 @{
    ViewBag.Title = "CreatePerson";
    Html.EnableClientValidation(false);
}
<h2>CreatePerson</h2>
@using (Html.BeginRouteForm("FormRoute", new {}, FormMethod.Post,
    new {@class = "personClass", data_formType = "person"}))
{
    <div class="dataElem">
        <label>PersonId</label>
        @Html.TextBoxFor(m => m.PersonId)
    </div>
    <div class="dataElem">
        <label>First Name</label>
        @Html.TextBoxFor(m => m.FirstName)
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        @Html.TextBoxFor(m => m.LastName)
    </div>
    <div class="dataElem">
        <label>Role</label>
        @Html.DropDownListFor(m => m.Role,
            new SelectList(Enum.GetNames(typeof (HelperMethods.Models.Role))))
    </div>
    <input type="submit" value="Submit" />
}
```

В него мы внесли одно изменение, которые выделено жирным шрифтом. По умолчанию, вспомогательные методы добавляют атрибуты `data` к HTML-элементам, таким образом обеспечивая валидацию форм, которую мы рассмотрели в главе 9 на примере приложения SportsStore. В этой главе мы не хотим их использовать, так что мы отключили их для представления `CreatePerson` с помощью метода `Html.EnableClientValidation`. Для остальной части приложения валидация на стороне клиента по-прежнему активирована. Подробно механизм валидации (в том числе и назначение `data`-атрибутов) мы рассмотрим в главе 23.

## Использование шаблонных вспомогательных методов

Для начала мы рассмотрим шаблонные вспомогательные методы `Html.Editor` и `Html.EditorFor`. Метод `Editor` принимает аргумент `string`, определяющий свойство, для которого требуется элемент `editor`; он проводит поиск соответствующего свойства в `ViewBag` и объекте модели (данний процесс был описан в главе 18).

Метод `EditorFor` является его строго типизированным эквивалентом, в котором свойство модели, необходимое для элемента `editor`, указывается с помощью лямбда-выражения.

В листинге 20-4 показано, как вспомогательные методы `Editor` и `EditorFor` используются в представлении `CreatePerson`. Как мы уже упоминали в главе 19, мы предпочитаем использовать строго типизированные вспомогательные методы, так как они снижают риск возникновения ошибки из-за опечатки в имени свойства; в этом листинге мы использовали оба типа, просто чтобы показать, что вы можете смешивать и сочетать методы так, как считаете нужным.

**Листинг 20-4:** Используем методы `Editor` и `EditorFor`

```
@model HelperMethods.Models.Person
@{
    ViewBag.Title = "CreatePerson";
    Html.EnableClientValidation(false);
}
<h2>CreatePerson</h2>
@using (Html.BeginRouteForm("FormRoute", new {}, FormMethod.Post,
    new {@class = "personClass", data_formType = "person"}))
{
    <div class="dataElem">
        <label>PersonId</label>
        @Html.Editor("PersonId")
    </div>
    <div class="dataElem">
        <label>First Name</label>
        @Html.Editor("FirstName")
    </div>
    <div class="dataElem">
        <label>Last Name</label>
        @Html.EditorFor(m => m.LastName)
    </div>
    <div class="dataElem">
        <label>Role</label>
        @Html.EditorFor(m => m.Role)
    </div>
    <div class="dataElem">
        <label>Birth Date</label>
        @Html.EditorFor(m => m.BirthDate)
    </div>
    <input type="submit" value="Submit" />
}
```

}

Editor и EditorFor создают идентичные элементы HTML; единственное различие между ними – это способ записи свойства, для которого создаются элементы editor. Вы можете увидеть результат применения этих методов, запустив приложение и перейдя по ссылке Home/CreatePerson, как показано на рисунке 20-1.

**Рисунок 20-1:** Используем методы Editor и EditorFor в форме

The screenshot shows a Windows-style window for 'CreatePerson'. Inside, there's a form with five fields: PersonId (100), First Name (Adam), Last Name (Freeman), Role (Admin), and Birth Date (01/01/0001 00:00:00). A 'Submit' button is at the bottom.

Не считая свойства BirthDate, эта форма по виду не отличается от формы, которую мы создали в главе 19. Тем не менее, изменения в ней довольно существенные; их можно увидеть, открыв страницу в другом браузере. На рисунке 20-2 показана та же страница в браузере Opera ([www.opera.com](http://www.opera.com)).

**Рисунок 20-2:** Отображение формы, созданной с помощью вспомогательных методов Editor и EditorFor

The screenshot shows the same 'CreatePerson' form in Opera. The Birth Date field is a date picker. A calendar overlay is displayed, showing the month of October 2012. The date '18' is highlighted in grey, indicating it is selected. Other dates are shown in red.

Обратите внимание, что элементы для свойств PersonId и BirthDate выглядят по-другому. У элемента PersonId появились стрелки (которые позволяют увеличивать и уменьшать значение) а элемент BirthDate теперь позволяет выбирать дату.

В спецификации HTML5 определены различные типы элементов `input`, которые используются для редактирования общих типов данных, таких как числа и даты. Методы `Editor` и `EditorFor` выбирают один из этих новых типов, основываясь на типе свойства, которое мы хотим отредактировать. В листинге 20-5 показан код HTML, который был создан для формы на рисунках 20-1 и 20-2.

**Листинг 20-5:** Элементы HTML `input`, созданные вспомогательными методами `Editor` и `EditorFor`

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>CreatePerson</title>
    <link href="/Content/Site.css" rel="stylesheet" />
    <style type="text/css">
        label {
            display: inline-block;
            width: 100px;
        }

        .dataElem {
            margin: 5px;
        }
    </style>
</head>
<body>
    <h2>CreatePerson</h2>
    <form action="/app/forms/Home/CreatePerson" class="personClass"
        data-formtype="person" method="post">
        <div class="dataElem">
            <label>PersonId</label>
            <input class="text-box single-line" id="PersonId" name="PersonId"
                type="number" value="0" />
        </div>
        <div class="dataElem">
            <label>First Name</label>
            <input class="text-box single-line" id="FirstName" name="FirstName"
                type="text" value="" />
        </div>
        <div class="dataElem">
            <label>Last Name</label>
            <input class="text-box single-line" id="LastName" name="LastName"
                type="text" value="" />
        </div>
        <div class="dataElem">
            <label>Role</label>
            <input class="text-box single-line" id="Role" name="Role"
                type="text" value="Admin" />
        </div>
        <div class="dataElem">
            <label>Birth Date</label>
            <input class="text-box single-line" id="BirthDate" name="BirthDate"
                type="datetime" value="01/01/0001 00:00:00" />
        </div>
        <input type="submit" value="Submit" />
    </form>
</body>
</html>
```

Атрибут `type` указывает, какой элемент `input` должен быть отображен. Для свойств `PersonId` and `BirthDate` во вспомогательных методах указаны типы `number` и `datetime`, для других свойств типом по умолчанию является `text`. Мы видим эти типы только в Opera по той причине, функции HTML5 все еще довольно новые и не все браузеры их поддерживают (в том числе Internet Explorer 10, который мы используем для скриншотов в этой книге).

Как видите, с помощью шаблонных вспомогательных методов мы смогли создать элементы формы, соответствующие контенту. Хотя в данном примере они были не особенно полезны, так как не все браузеры отображают типы элементов `input` HTML5, а также некоторые свойства, такие как `Role`, отображаются не так, как нам хотелось бы. Далее мы продемонстрируем, как обеспечить MVC Framework дополнительную информацию и улучшить код HTML, генерируемый вспомогательными методами. Но прежде чем мы углубимся в детали, давайте рассмотрим другие шаблонные вспомогательные методы. В таблице 20-1 приведен полный список вспомогательных методов, которые мы рассмотрим по отдельности в следующих разделах.

**Таблица 20-1:** Шаблонные вспомогательные методы MVC

Вспомогательный метод	Пример	Описание
Display	<code>Html.Display("FirstName")</code>	Визуализирует нередактируемый элемент HTML для указанного свойства модели в соответствии с типом свойства и метаданными.
DisplayFor	<code>Html.DisplayFor(x =&gt; x.FirstName)</code>	Строго типизированная версия предыдущего метода.
Editor	<code>Html.Editor("FirstName")</code>	Отображает элемент <code>editor</code> (редактируемый элемент HTML) для указанного свойства модели в соответствии с типом свойства и метаданными.
EditorFor	<code>Html.EditorFor (x =&gt; x.FirstName)</code>	Строго типизированная версия предыдущего метода.
Label	<code>Html.Label("FirstName")</code>	Отображает HTML-элемент <code>&lt;label&gt;</code> для указанного свойства модели
LabelFor	<code>Html.LabelFor (x =&gt; x.FirstName)</code>	Строго типизированная версия предыдущего метода.

## Создаем элементы `label` и `display`

Чтобы продемонстрировать другие вспомогательные методы, мы добавим в приложение новое представление, которое будет визуализировать нередактируемые элементы на основе данных, полученных из формы. Для начала мы обновили версию `HttpPost` действия `CreatePerson` контроллера `Home`, как показано в листинге 20-6.

**Листинг 20-6:** Указываем новое представление в контроллере `Home`

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

namespace HelperMethods.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewBag.Fruits = new string[] {"Apple", "Orange", "Pear"};
        }
    }
}
```

```

ViewBag.Cities = new string[] {"New York", "London", "Paris"};
string message = "This is an HTML element: <input>";
return View((object) message);
}

public ActionResult CreatePerson()
{
    return View(new Person {IsApproved = true});
}

[HttpPost]
public ActionResult CreatePerson(Person person)
{
    return View("DisplayPerson", person);
}
}
}
}

```

Мы создали представление `DisplayPerson.cshtml` в папке `/Views/Home`; содержимое этого файла показано в листинге 20-7.

#### **Листинг 20-7:** Содержимое файла `DisplayPerson.cshtml`

```

@model HelperMethods.Models.Person
 @{
    ViewBag.Title = "DisplayPerson";
}
<h2>DisplayPerson</h2>
<div class="dataElem">
    @Html.Label("PersonId")
    @Html.Display("PersonId")
</div>
<div class="dataElem">
    @Html.Label("FirstName")
    @Html.Display("FirstName")
</div>
<div class="dataElem">
    @Html.LabelFor(m => m.LastName)
    @Html.DisplayFor(m => m.LastName)
</div>
<div class="dataElem">
    @Html.LabelFor(m => m.Role)
    @Html.DisplayFor(m => m.Role)
</div>
<div class="dataElem">
    @Html.LabelFor(m => m.BirthDate)
    @Html.DisplayFor(m => m.BirthDate)
</div>

```

Чтобы увидеть вывод этого представления, запустите приложение, перейдите по ссылке `/Home/CreatePerson`, заполните форму и нажмите кнопку `Submit`. Результат показан на рисунке 20-3, и, как видите, мы сделали небольшой шаг назад, потому что вспомогательные методы `Label` и `LabelFor` используют имена свойств в качестве текста для меток.

**Рисунок 20-3:** Создаем нередактируемые элементы для объекта Person, используя вспомогательные методы



Вывод этих вспомогательных методов вы можете увидеть в листинге 20-8. Обратите внимание на то, что методы `Display` и `DisplayFor` не генерируют элементы HTML по умолчанию, они просто отображают значение свойства, с которым работают.

**Листинг 20-8:** HTML, созданный представлением `DisplayPerson`

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>DisplayPerson</title>
    <link href="/Content/Site.css" rel="stylesheet" />
    <style type="text/css">
        label {
            display: inline-block;
            width: 100px;
        }

        .dataElem {
            margin: 5px;
        }
    </style>
</head>
<body>
    <h2>DisplayPerson</h2>
    <div class="dataElem">
        <label for="PersonId">PersonId</label>
        100
    </div>
    <div class="dataElem">
        <label for="FirstName">FirstName</label>
        Adam
    </div>
    <div class="dataElem">
        <label for="LastName">LastName</label>
        Freeman
    </div>
    <div class="dataElem">
```

```

<label for="Role">Role</label>
    Admin
</div>
<div class="dataElem">
    <label for="BirthDate">BirthDate</label>
    01/01/0001 00:00:00
</div>
</body>
</html>

```

Хотя в данный момент польза от этих вспомогательных методов не очевидна, далее мы покажем вам, как можно изменить их поведение и сделать их вывод таким, каким вы хотели бы его видеть.

## Используем шаблонные вспомогательные методы для целой модели

Мы использовали шаблонные вспомогательные методы, которые генерируют вывод для одного свойства, но MVC Framework также определяет методы, которые работают с целыми объектами; этот процесс известен как *формирование шаблонов* (*scaffolding*). Вспомогательные методы для моделей показаны в таблице 20-2.

**Таблица 20-2:** Вспомогательные методы для моделей

Вспомогательный метод	Пример	Описание
DisplayForModel	Html.DisplayForModel()	Визуализирует нередактируемые элементы для объекта модели.
EditorForModel	Html.EditorForModel()	Визуализирует редактируемые элементы для объекта модели.
LabelForModel	Html.LabelForModel()	Визуализирует HTML-элемент <label> со ссылкой на объект модели.

В листинге 20-9 показано, как с помощью вспомогательных методов LabelForModel и EditorForModel можно упростить представление CreatePerson.cshtml.

**Листинг 20-9:** Применяем вспомогательные методы для моделей в представлении CreatePerson

```

@model HelperMethods.Models.Person
 @{
    ViewBag.Title = "CreatePerson";
    Html.EnableClientValidation(false);
}

<h2>CreatePerson: @Html.LabelForModel()</h2>

@using (Html.BeginRouteForm("FormRoute", new { }, FormMethod.Post,
    new { @class = "personClass", data_formType = "person" }))
{
    @Html.EditorForModel()
    <input type="submit" value="Submit" />
}

```

Результат применения вспомогательных методов для моделей показан на рисунке 20-4. Напомним еще раз, что пока вывод вспомогательных методов не вполне соответствует нашим ожиданиям. Метод LabelForModel сгенерировал не очень хорошие метки; хотя сейчас отображается больше свойств объекта модели Person, чем мы определили вручную в предыдущих примерах, но отображаются они не все (например, нет свойства Address) и не лучшим способом (например, свойство Role лучше отображать как элемент select, а не input).

**Рисунок 20-4:** Создаем редактируемые элементы для объекта модели Person с помощью вспомогательных методов для моделей

The screenshot shows a Microsoft Internet Explorer window with the title "CreatePerson". The address bar displays the URL "http://localhost:57520/Home/CreatePerson". The main content area is titled "CreatePerson:". It contains several input fields: "PersonId" with value "0", "FirstName" (empty), "LastName" (empty), "BirthDate" with value "01/01/0001 00:00:00", "IsApproved" (checked), and "Role" with value "Admin". Below the role input is a "Submit" button.

Часто проблемы заключаются в том, что сгенерированный этими методами HTML не соответствует стилям CSS, которые мы добавили в файл /Views/Shared/\_Layout.cshtml в главе 19. Ниже приведен пример кода HTML, сгенерированного для свойства FirstName:

```
<div class="editor-label">
    <label for="FirstName">FirstName</label>
</div>
<div class="editor-field">
    <input class="text-box single-line" id="FirstName" name="FirstName"
        type="text" value="" />
</div>
```

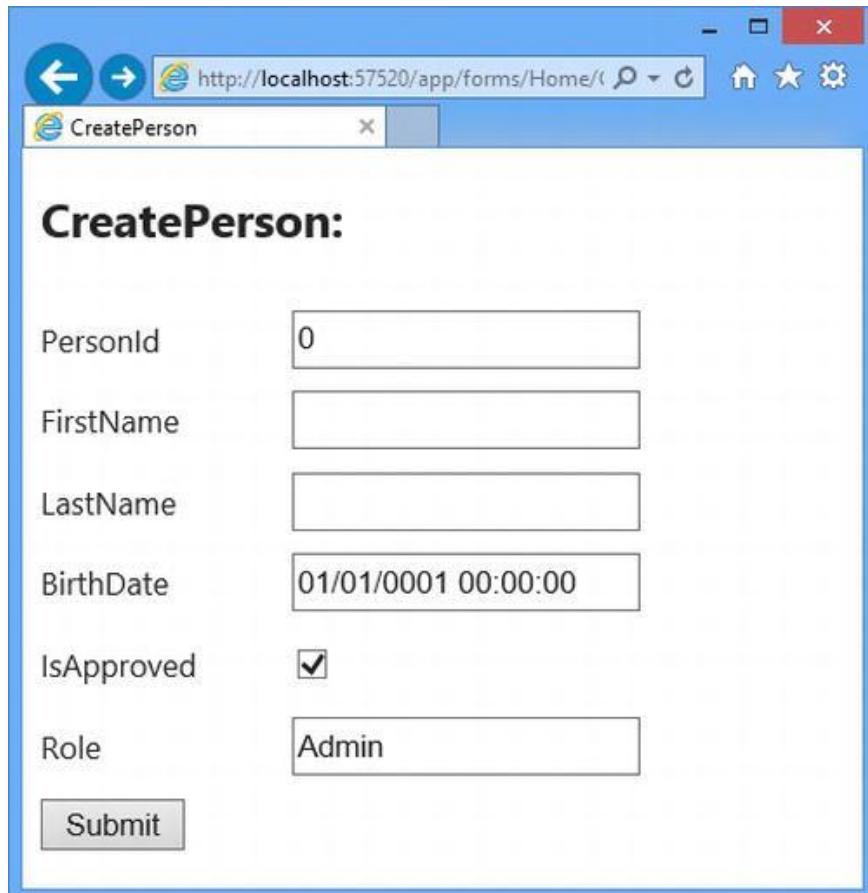
Чтобы привести в порядок наше представление, мы изменим стили так, чтобы они соответствовали значениям атрибутов CSS `class`, которые были добавлены к элементам `div` и `input` вспомогательными методами. В листинге 20-10 показаны изменения, которые мы внесли в файл \_Layout.cshtml.

**Листинг 20-10:** Изменяем стили CSS, определенные в файле \_Layout.cshtml

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
<meta name="viewport" content="width=device-width" />
<title>@ViewBag.Title</title>
<link href="~/Content/Site.css" rel="stylesheet"/>
<style type="text/css">
    label { display: inline-block; width: 100px; }
    .dataElem { margin: 5px; }
    h2 > label { width: inherit; }
    .editor-label, .editor-field { float: left; }
    .editor-label, .editor-label label, .editor-field input { height: 20px; }
    .editor-label { clear: left; }
    .editor-field { margin-left: 10px; margin-top: 10px; }
    input[type=submit] { float: left; clear: both; margin-top: 10px; }
    .column { float: left; margin: 10px; }
</style>
</head>
<body>
    @RenderBody()
</body>
</html>
```

Эти новые стили приводят страницу в соответствие с макетом, который мы использовали в предыдущих примерах, как показано на рисунке 20-5. (Мы также добавили некоторые стили, которые понадобятся нам позже в этой главе).

**Рисунок 20-5:** Применяем стили к элементам, используя классы, определенные вспомогательными методами



# Использование метаданных модели

Как вы уже убедились, шаблонные вспомогательные методы ничего не знают о нашем приложении и его типах моделей, и поэтому мы в конечном итоге получаем не тот HTML, который требуется. Мы хотим сохранить преимущества простых представлений, но мы должны улучшить качество вывода вспомогательных методов, прежде чем начнем использовать их всерьез.

Нельзя взваливать вину за такой результат на шаблонные вспомогательные методы; они генерируют HTML, основываясь на наиболее точных предположениях относительно того, что мы ожидаем. К счастью, с помощью метаданных модели мы можем предоставить вспомогательным методам информацию о том, как обрабатывать наши типы моделей. Метаданные записываются с помощью атрибутов C#, где значения атрибутов и параметров предоставляют различные инструкции вспомогательным методам представлений. Метаданные применяются к классу модели, к которому обращаются вспомогательные методы, когда генерируют элементы HTML. В следующих разделах мы продемонстрируем, как с помощью метаданных можно предоставлять инструкции вспомогательным методам для создания элементов `label`, `display` и `editor`.

## Контролируем редактируемость и видимость свойств с помощью метаданных

Мы не хотим, чтобы пользователи могли видеть или редактировать свойство `PersonId` класса `Person`. В большинстве классов моделей есть по крайней мере одно такое свойство, которое часто связано с механизмом хранения – например, первичный ключ, который находится под контролем реляционной базы данных (что мы продемонстрировали, когда создавали приложение `SportsStore`).

Можно использовать атрибут `HiddenInput`, который сообщает вспомогательному методу, что необходимо отобразить как скрытое поле. В листинге 20-11 показано, как мы применили `HiddenInputAttribute` к классу `Person`.

### Листинг 20-11: Используем атрибут `HiddenInput`

```
using System;
using System.Web.Mvc;

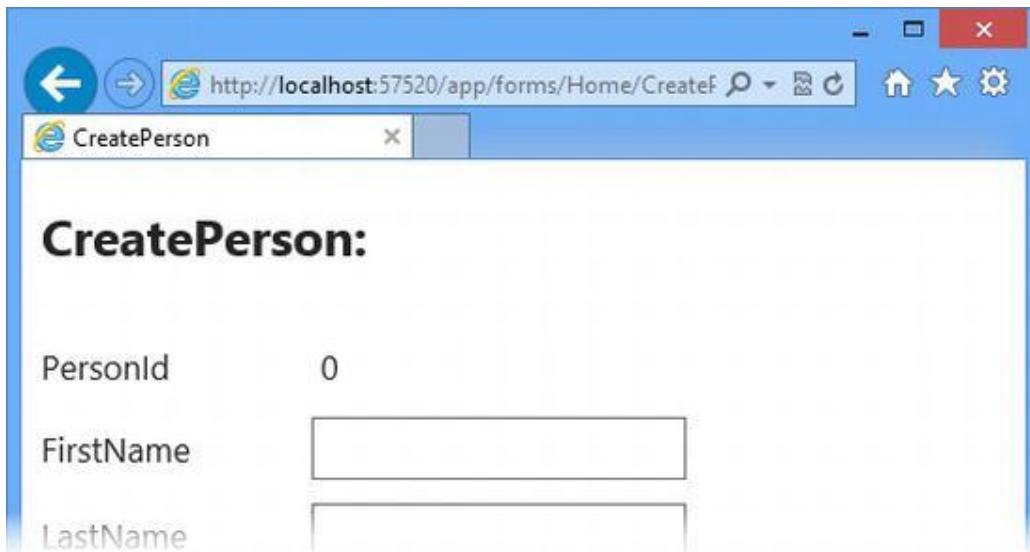
namespace HelperMethods.Models
{
    public class Person
    {
        [HiddenInput]
        public int PersonId { get; set; }

        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime BirthDate { get; set; }
        public Address HomeAddress { get; set; }
        public bool IsApproved { get; set; }
        public Role Role { get; set; }
    }

    // ...other types omitted from Listing 20-for brevity...
}
```

Когда к свойству применен этот атрибут, вспомогательные методы `Html.EditorFor` и `Html.EditorForModel` будут визуализировать для него нередактируемый элемент. На рисунке 20-6 показан результат запуска приложения и перехода по ссылке к `/Home/CreatePerson`.

**Рисунок 20-6:** Визуализация нередактируемого элемента для свойства



Значение свойства `PersonId` выводится, но пользователь не может его редактировать. Для этого свойства генерируется следующий HTML:

```
<div class="editor-field">
  0
  <input id="PersonId" name="PersonId" type="hidden" value="0" />
</div>
```

Значение свойства (в данном случае 0) просто визуализируется, но вспомогательный метод также создает для него скрытый элемент `input`; это полезный элемент в формах HTML, потому что он гарантирует, что при отправке формы мы предоставляем значение для данного свойства (мы вернемся к этой теме, когда будем рассматривать связывание данных в главе 22 и валидацию в главе 23). Если мы хотим полностью скрыть свойство, можно установить свойству `DisplayValue` в атрибуте `HiddenInput` значение `false`, как показано в листинге 20-12.

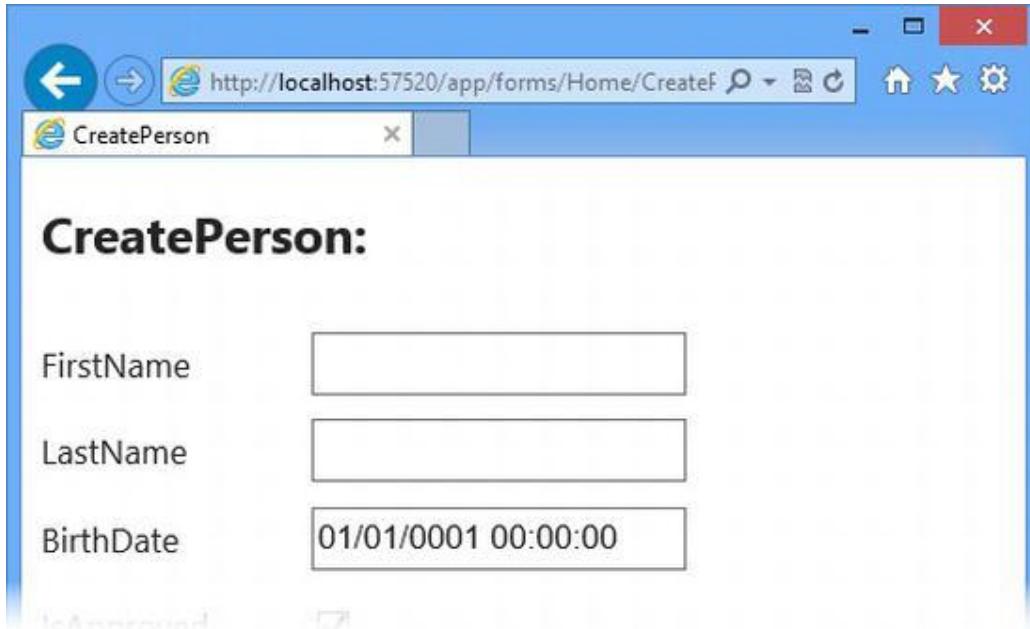
**Листинг 20-12:** Скрываем свойство с помощью атрибута `HiddenInput`

```
public class Person
{
  [HiddenInput(DisplayValue = false)]
  public int PersonId { get; set; }

  public string FirstName { get; set; }
  public string LastName { get; set; }
  public DateTime BirthDate { get; set; }
  public Address HomeAddress { get; set; }
  public bool IsApproved { get; set; }
  public Role Role { get; set; }
}
```

Когда мы используем вспомогательный метод `Html.EditorForModel` для объекте `Person`, он создает скрытое поле ввода, чтобы при отправке формы значение свойства `PersonId` также было отправлено, но метка и числовое значение были опущены. Это имеет такой же эффект, как и скрытие свойства `PersonId` от пользователя, как показано на рисунке 20-7.

**Рисунок 20-7:** Скрытие свойств объекта модели от пользователя



Если вы хотите визуализировать HTML для отдельных свойств, можно создать скрытый элемент `input` для свойства `PersonId` с помощью вспомогательного метода `Html.EditorFor`:

```
@Html.EditorFor(m => m.PersonId)
```

Свойство `HiddenInput` обнаруживается, и если `DisplayValue` имеет значение `true`, то будет сгенерирован следующий код HTML:

```
<input id="PersonId" name="PersonId" type="hidden" value="1" />
```

### Исключаем свойство из формирования шаблонов

Если вы не хотите создавать HTML для свойства, можно использовать атрибут `ScaffoldColumn`. В то время как атрибут `HiddenInput` включает значение свойства в скрытый элемент ввода, `ScaffoldColumn` означает, что свойство не будет использоваться при формировании шаблона. Вот пример использования атрибута:

```
[ScaffoldColumn(false)]  
public int PersonId { get; set; }
```

Когда вспомогательный метод для модели увидит атрибут `ScaffoldColumn`, он полностью пропустит это свойство; для него не будет создан скрытый элемент ввода, и информация из этого свойства не будет включена в HTML. Сгенерированный HTML будет таким же, как если бы мы использовали атрибут `HiddenInput`, но при отправке формы значение для этого свойства отправляться не будет (это может оказывать влияние на связывание данных, которое мы обсудим позже в этой главе).

Атрибут `ScaffoldColumn` не влияет на вспомогательные методы, работающие с одним свойством, такие как `EditorFor`. Если мы вызовем в представлении `@Html.EditorFor(m => m.PersonId)`, для свойства `PersonId` будет создан элемент `editor`, даже если к нему применен атрибут `ScaffoldColumn`.

## Используем метаданные для создания элементов label

По умолчанию вспомогательные методы Label, LabelFor, LabelForModel и EditorForModel используют имена свойств как содержимое для элементов label, которые они генерируют. Например, если мы визуализируем метку таким образом:

```
@Html.LabelFor(m => m.BirthDate)
```

генерируется следующий элемент HTML:

```
<label for="BirthDate">BirthDate</label>
```

Конечно, мы не всегда хотим отображать пользователю названия свойств. Для этого можно применить атрибут DisplayName из пространства имен System.ComponentModel.DataAnnotations и передать в него значение для свойства Name. В листинге 20-13 показано, как мы применили этот атрибут к классу Person.

**Листинг 20-13:** Используем атрибут DisplayName для создания метки

```
using System;
using System.Web.Mvc;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel;

namespace HelperMethods.Models
{
    [DisplayName("New Person")]
    public class Person
    {
        [HiddenInput(DisplayValue = false)]
        public int PersonId { get; set; }

        [Display(Name = "First")]
        public string FirstName { get; set; }

        [Display(Name = "Last")]
        public string LastName { get; set; }

        [Display(Name = "Birth Date")]
        public DateTime BirthDate { get; set; }

        public Address HomeAddress { get; set; }

        [Display(Name = "Approved")]
        public bool IsApproved { get; set; }

        public Role Role { get; set; }
    }

    // ...other types omitted from Listing 20-for brevity...
}
```

Когда вспомогательный метод будет визуализировать элемент label для свойства BirthDate, он обнаружит атрибут Display и будет использовать значение параметра Name в качестве текста метки, например:

```
<label for="BirthDate">Birth Date</label>
```

Вспомогательные методы также распознают атрибут DisplayName, который можно найти в пространстве имен System.ComponentModel. Этот атрибут можно применять к классам, что

позволяет нам использовать вспомогательный метод `Html.LabelForModel` – в листинге показано, как мы применили этот атрибут к классу `Person`. (Атрибут `DisplayName` можно применить и к свойствам, но мы, как правило, используем его только для классов моделей – просто по привычке). Результат применения атрибутов `Display` и `DisplayName` показан на рисунке 20-8.

**Рисунок 20-8:** Используем атрибуты `Display` и `DisplayName` для создания меток

The screenshot shows a web browser window with the URL `http://localhost:57520/app/forms/Home/CreatePerson`. The page title is "CreatePerson: New Person". The form contains five fields: "First" (text input), "Last" (text input), "Birth Date" (text input with value "01/01/0001 00:00:00"), "Approved" (checkbox checked), and "Role" (text input with value "Admin"). Below the form is a "Submit" button.

### Используем метаданные для значений данных

С помощью метаданных можно предоставить инструкции относительно того, как должно отображаться свойство модели. Для нашего свойства `BirthDate` отображается время, хотя мы хотели бы отображать для него только дату, и, чтобы решить эту проблему, мы будем использовать метаданные. Отображаемые значения данных контролируются атрибутом `DataType`. В листинге 20-14 вы можете увидеть, как мы применили его к классу `Person`.

**Листинг 20-14:** Применяем атрибут `DataType` к классу `Person`

```
[DisplayName("New Person")]
public class Person
{
    [HiddenInput(DisplayValue = false)]
    public int PersonId { get; set; }

    [Display(Name = "First")]
    public string FirstName { get; set; }

    [Display(Name = "Last")]
    public string LastName { get; set; }

    [Display(Name = "Birth Date")]
    [DataType(DataType.Date)]
```

```

public DateTime BirthDate { get; set; }

public Address HomeAddress { get; set; }

[Display(Name = "Approved")]
public bool IsApproved { get; set; }

public Role Role { get; set; }
}

```

Атрибут `DataType` принимает в качестве параметра значение из перечисления `DataType`. В данном примере мы указали значение `DateTime.Date`, которое сообщает шаблонным вспомогательным методам, что необходимо визуализировать значение свойства `BirthDate` в формате даты без времени, как показано на рисунке 20-9.

#### *Подсказка*

*Изменения будут более заметны, если вы просмотрите приложение в браузере с лучшей поддержкой типов элементов input HTML5.*

**Рисунок 20-9:** Изменяем способ отображения значения `DateTime` с помощью атрибута `DataType`

The screenshot shows a user interface with three form fields. The first field is labeled 'Last' and contains an empty text input box. The second field is labeled 'Birth Date' and contains the value '01/01/0001' in a text input box. The third field is labeled 'Approved' and has a checked checkbox. Below the form, there are two buttons: 'Previous' and 'Admin'.

В таблице 20-3 описаны наиболее полезные значения из перечисления `DataType`.

**Таблица 20-3:** Значения из перечисления `DataType`

Значение	Описание
<code>DateTime</code>	Отображает дату и время (это поведение по умолчанию для значений <code>System.DateTime</code> )
<code>Date</code>	Отображает дату из <code>DateTime</code>
<code>Time</code>	Отображает время из <code>DateTime</code>
<code>Text</code>	Отображает одну строку текста
<code>PhoneNumber</code>	Отображает номер телефона
<code>MultilineText</code>	Визуализирует значение в элементе <code>textarea</code>
<code>Password</code>	Отображает замаскированные символы
<code>Url</code>	Отображает данные в виде URL (используя элемент HTML <code>a</code> )
<code>EmailAddress</code>	Отображает данные как адрес электронной почты (используя элемент <code>a</code> с <code>mailto href</code> )

Результат применения этих значений зависит от типа свойства, с которым они связаны, и вспомогательного метода, который мы используем. Например, значение `MultilineText` сообщает вспомогательному методу, который создает элементы `editor` для свойств, сгенерировать элемент

HTML `textarea`, но он будет проигнорирован вспомогательными методами для элементов `display`. Все логично - элемент `textarea` позволяет пользователю редактировать значение, и атрибут никак не повлияет на данные, которые мы отображаем в формате *read-only*. Равным образом, значение URL будет влиять только на вспомогательные методы для элементов `display`, которые будут визуализировать элемент HTML `a` для создания ссылки.

## Используем метаданные для выбора шаблона отображения

Как следует из их названия, шаблонные вспомогательные методы используют *шаблоны отображения* (*display templates*) для создания HTML. Шаблон выбирается на основании типа обрабатываемого свойства и самого вспомогательного метода. С помощью атрибута `UIHint` можно указать шаблон, который мы хотим использовать для работы с определенным свойством, как показано в листинге 20-15.

**Листинг 20-15:** Используем атрибут `UIHint`

```
[DisplayName("New Person")]
public class Person
{
    [HiddenInput(DisplayValue = false)]
    public int PersonId { get; set; }

    [Display(Name = "First")]
    [UIHint("MultilineText")]
    public string FirstName { get; set; }

    [Display(Name = "Last")]
    public string LastName { get; set; }

    [Display(Name = "Birth Date")]
    [DataType(DataType.Date)]
    public DateTime BirthDate { get; set; }

    public Address HomeAddress { get; set; }

    [Display(Name = "Approved")]
    public bool IsApproved { get; set; }

    public Role Role { get; set; }
}
```

В листинге мы указали шаблон `MultilineText`, который создаст элемент HTML `textarea` для свойства `FirstName`; он используется с вспомогательными методами для элементов `editor`, такими как `EditorFor` или `EditorForModel`. В таблице 20-4 показан набор встроенных шаблонов MVC Framework.

**Таблица 20-4:** Встроенные шаблоны MVC Framework

Название	Эффект ( <code>Editor</code> )	Эффект ( <code>Display</code> )
Boolean	Отображает чекбокс для свойств <code>bool</code> . Для свойств <code>bool</code> , поддерживающих значение <code>null</code> , создается элемент <code>select</code> с опциями <code>True</code> , <code>False</code> и <code>Not Set</code> .	Как и для элементов <code>editor</code> , но добавляется атрибут <code>disabled</code> , который визуализирует элементы управления HTML в формате <code>read-only</code> .
Collection	Отображает соответствующий шаблон для каждого элемента в последовательности <code>IEnumerable</code> . Элементы в последовательности	Как и для элементов <code>editor</code> .

Название	Эффект (Editor)	Эффект (Display)
	не обязательно должны быть одного типа.	
Decimal	Отображает однострочное текстовое поле и приводит значения данных к формату двух десятичных разрядов.	Отображает значения данных в формате двух десятичных разрядов.
DateTime	Визуализирует элемент <code>input</code> , атрибут <code>type</code> которого содержит <code>datetime</code> (позволяет ввести дату и время).	Визуализирует значение переменной <code>DateTime</code> .
Date	Визуализирует элемент <code>input</code> , атрибут <code>type</code> которого содержит <code>date</code> (позволяет ввести дату, но не время).	Визуализирует дату из переменной <code>DateTime</code>
EmailAddress	Визуализирует значение в однострочном элементе ввода <code>textbox</code> .	Отображает ссылку с помощью элемента HTML <code>a</code> с атрибутом <code>href="mailto:</code>
HiddenInput	Создает скрытый элемент ввода.	Отображает значение данных и создает скрытый элемент ввода.
Html	Отображает значение в однострочном элементе ввода <code>textbox</code> .	Отображает ссылку с помощью элемента HTML <code>a</code> .
MultilineText	Отображает элемент HTML <code>textarea</code> , который содержит значения данных.	Отображает значения данных.
Number	Отображает элемент ввода, атрибут <code>type</code> которого содержит <code>number</code> .	Отображает значения данных.
Object	Объяснение дано после этой таблицы.	Объяснение дано после этой таблицы.
Password	Отображает значение в элементе ввода <code>textbox</code> таким образом, что символы замаскированы, но могут быть отредактированы.	Отображает значения данных, символы не замаскированы.
String	Отображает значение в однострочном элементе ввода <code>textbox</code> .	Отображает значения данных.
Text	Идентичен шаблону <code>String</code> .	Идентичен шаблону <code>String</code> .
Tel	Отображает элемент ввода, атрибут <code>type</code> которого содержит <code>tel</code> .	Отображает значения данных.
Time	Отображает элемент ввода, атрибут <code>type</code> которого содержит <code>time</code> (позволяет установить время, но не дату).	Отображает дату из переменной <code>DateTime</code>
Url	Отображает значение в элементе ввода <code>textbox</code> .	Отображает ссылку с помощью элемента HTML <code>a</code> . Для значений данных устанавливается атрибут <code>href</code> и внутренний HTML.

#### Внимание!

*Используя атрибут `UIHint`, будьте внимательны. Если мы выберем для свойства шаблон, который не может работать с его типом, то получим исключение: например, применив шаблон `Boolean` к свойству `string`.*

Шаблон `Object` является особым случаем. Он используется шаблонными вспомогательными методами, чтобы создать HTML для объекта модели представления. Этот шаблон проверяет каждое свойство объекта и выбирает наиболее подходящий шаблон для типа каждого свойства. Шаблон `Object` учитывает метаданные, такие как атрибуты `UIHint` и `DataType`.

## Применяем метаданные в дополняющем классе (buddy class)

Не всегда можно применить метаданные к классу сущности модели. Так обычно происходит, когда классы моделей генерируются автоматически, например, с помощью инструментов ORM, таких как Entity Framework (хотя и не таким образом, как мы использовали Entity Framework в приложении SportsStore). Любые изменения, которые мы вносим в автоматически сгенерированные классы (такие как применение атрибутов), будут потеряны при следующем обновлении или регенерации классов.

Чтобы решить эту проблему, нужно определить класс модели как частичный и создать еще один частичный класс, который содержит метаданные. Многие инструменты для автоматической генерации классов создают частичные классы по умолчанию, в том числе и Entity Framework. В листинге 20-16 показан измененный класс `Person`, который мог бы быть сгенерированным автоматически: в нем нет метаданных, и он определен как частичный.

### Листинг 20-16: Частичный класс модели

```
using System;
using System.ComponentModel.DataAnnotations;

namespace HelperMethods.Models
{
    [MetadataType(typeof(PersonMetadata))]
    public partial class Person
    {
        public int PersonId { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime BirthDate { get; set; }
        public Address HomeAddress { get; set; }
        public bool IsApproved { get; set; }
        public Role Role { get; set; }
    }

    // ...other types omitted from listing for brevity...
}
```

Мы сообщаем MVC Framework о дополняющем классе с помощью атрибута `MetadataType`, который принимает тип дополняющего класса в качестве аргумента. Дополняющий класс должен быть определен в том же пространстве имен и также должен быть частичным. Чтобы продемонстрировать, как это работает, мы добавили в проект новую папку под названием `Models/Metadata`. В этой папке мы создали новый класс под названием `PersonMetadata.cs`, содержание которого показано в листинге 20-17.

### Листинг 20-17: Определяем дополняющий класс метаданных

```
using System;
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;
using System.Web.Mvc;

namespace HelperMethods.Models
{
    [DisplayName("New Person")]
    public partial class PersonMetadata
```

```

{
    [HiddenInput(DisplayValue = false)]
    public int PersonId { get; set; }

    [Display(Name = "First")]
    public string FirstName { get; set; }

    [Display(Name = "Last")]
    public string LastName { get; set; }

    [Display(Name = "Birth Date")]
    [DataType(DataType.Date)]
    public DateTime BirthDate { get; set; }

    [Display(Name = "Approved")]
    public bool IsApproved { get; set; }
}
}

```

Дополняющий класс должен содержать только те свойства, к которым мы хотим применить метаданные – нет необходимости воспроизводить все свойства класса `Person`, например.

#### *Подсказка*

*Обязательно измените пространство имен, в которое Visual Studio добавляет новый класс - дополняющий класс должен быть в том же пространстве имен, что и класс модели (в данном примере - `HelperMethods.Models`).*

---

## Работаем со сложными типами свойств

Процесс формирования шаблонов полагается на шаблон `Object`, который мы описали в предыдущем разделе. Каждое свойство проверяется, и для него выбирается шаблон для создания кода HTML, который будет визуализировать это свойство и его значения данных.

Возможно, вы заметили, что свойство `HomeAddress` не отображалось как часть класса `Person`, когда мы использовали `EditorForModel`. Так происходит потому, что шаблон `Object` работает только с простыми типами, то есть типами, которые могут быть получены из значения `string` с помощью метода `GetConverter` класса `System.ComponentModel.TypeDescriptor`. Поддерживаемые типы включают внутренние типы C#, такие как `int`, `bool` и `double`, а также многие общие типы MVC Framework, такие как `Guid` и `DateTime`.

Из этого следует, что шаблонные вспомогательные методы не являются рекурсивными. Получив объект для обработки, шаблонный вспомогательный метод генерирует HTML только для простых типов свойств и проигнорирует все свойства, которые представляют собой сложные объекты.

Возможно, это и неудобно, но это разумная политика; MVC Framework не знает, как были созданы наши объекты моделей, и если бы шаблон `Object` был рекурсивным, то он бы задействовал медленно загружающиеся функции ORM, а затем читал и отображал бы каждый объект в нижележащей базе данных. Если мы хотим создать HTML для сложного свойства, то это нужно сделать явно, используя отдельный вызов к шаблонному вспомогательному методу. Чтобы показать, как это делается, мы внесли изменения в представление `CreatePerson.cshtml`, которые показаны в листинге 20-18.

#### Листинг 20-18: Работаем со сложным типом

```
@model HelperMethods.Models.Person
```

```

@{
    ViewBag.Title = "CreatePerson";
    Html.EnableClientValidation(false);
}

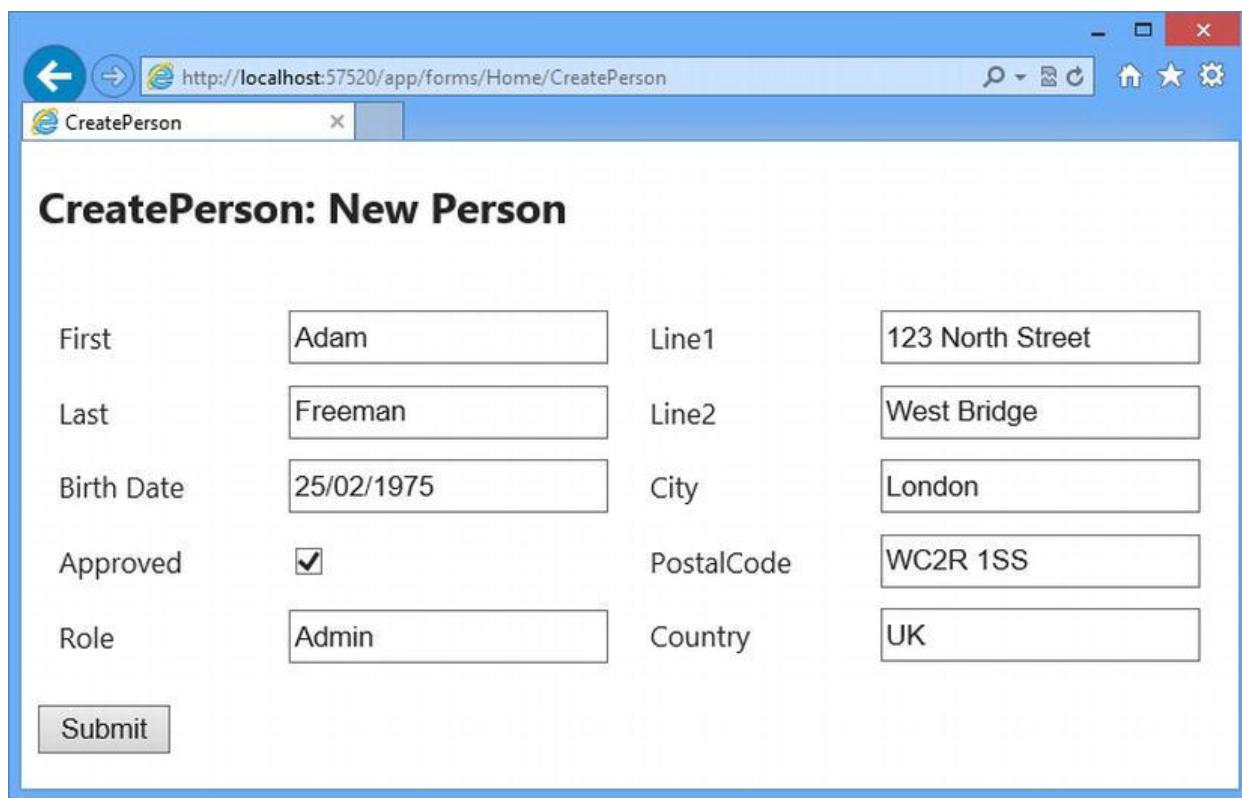
<h2>CreatePerson: @Html.LabelForModel()</h2>

@using (Html.BeginRouteForm("FormRoute", new {}, FormMethod.Post,
    new {@class = "personClass", data_formType = "person"}))
{
    <div class="column">
        @Html.EditorForModel()
    </div>
    <div class="column">
        @Html.EditorFor(m => m.HomeAddress)
    </div>
    <input type="submit" value="Submit" />
}

```

Чтобы отобразить свойство `HomeAddress`, мы добавили вызов к строго типизированному вспомогательному методу `EditorFor`. (Мы также добавили некоторые элементы `div`, чтобы к генерируемому HTML применялись стили CSS, которые мы определили для класса `column` в листинге 20-10). Результат показан на рисунке 20-10.

**Рисунок 20-10:** Отображаем сложное свойство



#### Подсказка

*Свойство `HomeAddress` возвращает объект `Address`, и мы можем применить к классу `Address` те же метаданные, которые применили к классу `Person`. Когда мы используем вспомогательный метод `EditorFor` в свойстве `HomeAddress`, шаблон `Object` вызывается явно, так что все соглашения, относящиеся к метаданным, соблюdenы.*

# Настройка системы шаблонных вспомогательных методов

Мы научились с помощью метаданных влиять на то, как шаблонные вспомогательные методы отображают элементы. Однако, как и для всех функций MVC Framework, для шаблонных вспомогательных методов есть дополнительные опции, которые позволяют полностью настроить их работу. В следующих разделах мы продемонстрируем, как можно дополнить или заменить встроенную поддержку для реализации конкретных решений.

## Создаем пользовательский шаблон Editor

Самый простой способ изменить поведение шаблонного вспомогательного метода – создать пользовательский шаблон. Это позволит нам сгенерировать для свойства модели именно такой HTML, какой мы хотим.

Чтобы продемонстрировать, как работает этот подход, мы создадим пользовательский шаблон для свойства `Role` класса `Person`. Тип этого свойства – значение из перечисления `Role`, и нам не нравится то, как оно визуализируется. Шаблонный вспомогательный метод создает обычный элемент `input`, который позволяет пользователю ввести любое значение, не ограничивая их теми, которые указаны в перечислении.

MVC Framework будет искать пользовательские шаблоны для элементов `editor` в папке `/Views/Shared/EditorTemplates`, поэтому мы добавили эту папку в проект и создали в ней новое строго типизированное частичное представление под названием `Role.cshtml`. Вы можете увидеть содержимое этого файла в листинге 20-19.

### Листинг 20-19: Содержимое файла Role.cshtml

```
@model HelperMethods.Models.Role  
  
@Html.DropDownListFor(m => m,  
    new SelectList(Enum.GetNames(Model.GetType()),  
        Model.ToString()))
```

Тип модели для данного представления – это перечисление `Role`, и мы используем вспомогательный метод `Html.DropDownListFor`, чтобы создать элемент `select` с элементами `option` для значений в перечислении. Мы передаем дополнительное значение в конструктор `SelectList`; оно указывает выбранное значение, которое мы получаем от объекта модели представления. Метод `DropDownListFor` и объект `SelectList` работают со строковыми значениями, поэтому мы должны преобразовать значения в перечислении и модели представления.

Когда мы будем использовать шаблонный вспомогательный метод, чтобы создать элемент `editor` для типа `Role`, будет использоваться файл `/Views/Shared/EditorTemplates/Role.cshtml`. Таким образом, мы получаем последовательное и полезное представление для типа данных. Эффект применения пользовательского шаблона показан на рисунке 20-11.

### Рисунок 20-11: Эффект применения пользовательского шаблона для перечисления Role



The screenshot shows a portion of a web application's edit form. On the left, there are labels for 'Birth Date' (with a text input containing '01/01/0001'), 'Approved' (with a checkbox checked), and 'Role' (with a dropdown menu). The 'Role' dropdown menu is open, showing three options: 'Admin', 'User', and 'Guest'. The 'Guest' option is highlighted with a blue background and white text. To the right of the dropdown are other form fields: 'City' (empty), 'PostalCode' (empty), and 'Country' (empty). At the bottom left is a 'Submit' button.

## Порядок поиска шаблонов

Шаблон `Role.cshtml` работает потому, что платформа MVC сначала выполняет поиск пользовательских шаблонов для данного типа C#, и только потом использует один из встроенных шаблонов. Поиск подходящего шаблона MVC Framework проводит в следующем порядке:

1. Шаблон, переданный во вспомогательный метод - например, вспомогательный метод `Html.EditorFor(m => m.SomeProperty, "MyTemplate")` нашел бы шаблон `MyTemplate`.
2. Шаблон, который указан в атрибутах метаданных, таких как `UIHint`.
3. Шаблон, который связан с типом данных, указанным в метаданных, например в атрибуте `DataType`.
4. Шаблон, который соответствует имени класса .NET обрабатываемого типа данных.
5. Встроенный шаблон `String`, если обрабатываемый тип данных является простым.
6. Шаблон, который соответствует базовому классу типа данных.
7. Если тип данных реализует `IEnumerable`, то будет использоваться встроенный шаблон `Collection`. Если ничего не подходит, то будет использоваться шаблон `Object`, на который распространяется правило, что формирование шаблонов не является рекурсивным.

В некоторых пунктах используются встроенные шаблоны, которые описаны в таблице 20-4. На каждом этапе процесса поиска шаблона MVC Framework ищет шаблон под названием `EditorTemplates/<name>` для вспомогательных методов для элементов `editor` и `DisplayTemplates/<name>` для вспомогательных методов для элементов `display`. Для нашего шаблона `Role` поиск остановится на пункте 4; мы создали шаблон под названием `Role.cshtml` и поместили его в папку `/Views/Shared/EditorTemplates`.

Поиск пользовательских шаблонов проводится по той же схеме, что и поиск обычных представлений, а следовательно, мы можем создать пользовательский шаблон для конкретного контроллера и поместить его в папку `~/Views/<controller>/EditorTemplates`, чтобы переопределить шаблоны из папки `~/Views/Shared`. Подробная информация о поиске представлений дана в главе 18.

## Создаем общий (generic) шаблон

Можно создавать шаблоны не только для конкретного типа. Мы можем, например, создать шаблон для всех перечислений, а затем выбрать его с помощью атрибута `UIHint`. Если вы посмотрите блок «Порядок поиска шаблонов», то увидите, что шаблоны, заданные с помощью атрибута `UIHint`, имеют приоритет над шаблонами для конкретного типа.

Чтобы продемонстрировать, как это работает, мы создали новое представление под названием `Enum.cshtml` в папке `/Views/Shared/EditorTemplates`. Содержимое этого файла показано в листинге 20-20.

### Листинг 20-20: Представление `Enum.cshtml`

```
@model Enum
@Html.DropDownListFor(m => m, Enum.GetValues(Model.GetType()))
    .Cast<enum>()
    .Select(m =>
{
    string enumVal = Enum.GetName(Model.GetType(), m);
    return new SelectListItem()
    {
        Selected = (Model.ToString() == enumVal),
```

```

        Text = enumVal,
        Value = enumVal
    };
}

```

Типом модели для этого шаблона является `Enum`, что позволяет нам работать с любым перечислением. Для разнообразия мы создали с помощью LINQ строки, которые необходимы для создания элементов `select` и `option`. Далее мы можем затем применить атрибут `UIHint`. В нашем проекте есть дополняющий класс метаданных, так что мы применили этот атрибут к классу `PersonMetadata`, как показано в листинге 20-21. (Напомним, что этот класс находится по адресу `/Models/Metadata/PersonMetadata.cs`).

**Листинг 20-21:** Используем атрибут `UIHint`, чтобы указать пользовательский шаблон

```

using System;
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;
using System.Web.Mvc;

namespace HelperMethods.Models
{
    [DisplayName("New Person")]
    public partial class PersonMetaDataTable
    {
        [HiddenInput(DisplayValue = false)]
        public int PersonId { get; set; }

        [Display(Name = "First")]
        public string FirstName { get; set; }

        [Display(Name = "Last")]
        public string LastName { get; set; }

        [Display(Name = "Birth Date")]
        [DataType(DataType.Date)]
        public DateTime BirthDate { get; set; }

        [Display(Name = "Approved")]
        public bool IsApproved { get; set; }

        [UIHint("Enum")]
        public Role Role { get; set; }
    }
}

```

Такой подход позволяет получить более общее решение, которое можно применять во всем приложении и гарантировать, что все свойства `Enum` отображаются с помощью элемента `select`. Мы предпочитаем создавать пользовательские шаблоны для конкретных типов моделей, но иногда удобнее иметь шаблон, который можно применять более широко.

## Заменяем встроенные шаблоны

Если мы создадим пользовательский шаблон с таким же именем, как и один из встроенных шаблонов, MVC Framework будет использовать пользовательскую версию в предпочтение встроенной. В листинге 20-22 показано содержимое файла `Boolean.cshtml`, который мы создали в папке `/Views/Shared/EditorTemplates`. Это представление заменяет встроенный шаблон `Boolean`, который используется для отображения значений `bool` и `bool?`.

## Листинг 20-22: Заменяем встроенный шаблон

```
@model bool?  
  
@if (ViewData.ModelMetadata.IsNullableValueType && Model == null) {  
    @:(True) (False) <b>(Not Set)</b>  
} else if (Model.Value) {  
    @:<b>(True)</b> (False) (Not Set)  
} else {  
    @:(True) <b>(False)</b> (Not Set)  
}
```

В данном представлении мы отображаем все возможные значения и выделяем то, которое соответствует объекту модели. Результат применения этого шаблона показан на рисунке 20-12.

**Рисунок 20-12:** Результат переопределения встроенного шаблона для элементов editor

Last		Line2	
Birth Date	01/01/0001	City	
Approved	(True) ( <b>False</b> ) (Not Set)	PostalCode	
Role	Guest ▾	Country	

Обратите внимание на гибкость пользовательских шаблонов, хотя в этом примере мы используем их не самым полезным образом и даже не позволяем пользователю изменять значение свойства. Как видите, существует много способов контролировать, как должны отображаться и редактироваться свойства модели, и вы можете выбрать подход, который лучше всего соответствует вашему стилю программирования и конкретному приложению.

## Резюме

В этой главе мы рассмотрели систему шаблонов моделей, которые доступны через шаблонные вспомогательные методы представлений. Хотя создание метаданных и пользовательских шаблонов может занять некоторое время, но с их помощью вы можете полностью настроить способ отображения и редактирования данных модели представления и получить наилучший результат для конкретного приложения.

# Вспомогательные методы для URL и Ajax

В этой главе мы собираемся закончить со вспомогательными методами MVC. Мы покажем вам те методы, которые способны генерировать URL-адреса, ссылки и элементы с включенным Ajax. Ajax является ключевой особенностью любого богатого веб-приложения, и MVC фреймворк включает в себя некоторые полезные функции, которые основаны на библиотеке JQuery. Мы покажем вам, как все это работает и как вы можете создавать формы и ссылки с включенным Ajax.

## Примечание

*Вам нужно будет чистить историю браузера, когда вы будете переходить в этой главе от одного примера к следующему: это потому что мы строим наши функции постепенно. В этой главе мы добавили напоминания по ключевым моментам, но если вы не получите ожидаемого результата от примера, прежде всего – очистите историю.*

---

## Обзор и подготовка проекта для примера

Мы собираемся продолжать использовать проект HelperMethods, который мы создали в главе 19 и дополнили в главе 20. Некоторые из наших примеров в данной главе будут использовать информацию о маршрутизации, которую мы определили в проекте, и это, как напоминание, показано в листинге 21-1.

**Листинг 21-1:** Содержание файла /App\_Start/RouteConfig.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;
namespace HelperMethods
{
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
            routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new
                {
                    controller = "Home",
                    action = "Index",
                    id = UrlParameter.Optional
                }
            );
            routes.MapRoute(
                name: "FormRoute",
                url: "app/forms/{controller}/{action}"
            );
        }
    }
}
```

Наша конфигурация маршрутизации довольно проста. Она состоит из роута по умолчанию, добавленного Visual Studio, когда проект был создан, и дополнительного роута `FormRoute`, который имеет два статических сегмента.

Для этой главы мы создали новый контроллер `People`, как показано в листинге 21-2. Этот контроллер определяет коллекцию объектов `Person`, которую мы будем использовать, чтобы продемонстрировать различные функции вспомогательных методов.

### Листинг 21-2: Контроллер `People`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using HelperMethods.Models;
namespace HelperMethods.Controllers
{
    public class PeopleController : Controller
    {
        private Person[] personData = {
            new Person {FirstName = "Adam", LastName = "Freeman", Role = Role.Admin},
            new Person {FirstName = "Steven", LastName = "Sanderson", Role = Role.Admin},
            new Person {FirstName = "Jacqui", LastName = "Griffyth", Role = Role.User},
            new Person {FirstName = "John", LastName = "Smith", Role = Role.User},
            new Person {FirstName = "Anne", LastName = "Jones", Role = Role.Guest}
        };

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

        public ActionResult GetPeople()
        {
            return View(personData);
        }

        [HttpPost]
        public ActionResult GetPeople(string selectedRole)
        {
            if (selectedRole == null || selectedRole == "All")
            {
                return View(personData);
            }
            else
            {
                Role selected = (Role)Enum.Parse(typeof(Role), selectedRole);
                return View(personData.Where(p => p.Role == selected));
            }
        }
    }
}
```

Мы не использовали никаких новых технических приемов в этом контроллере. Метод действия `Index` возвращает представление по умолчанию. Мы будем использовать два метода действия `GetPeople` для обработки простой формы. Новые возможности в этой главе появляются в представлениях, которые мы создадим, когда будем показывать различные вспомогательные методы.

Мы также должны добавить к проекту некоторые новые стили CSS. В предыдущих главах мы показали, что можно определить стили в отдельных представлениях или макетах, но мы собираемся

определить нужные нам стили в файле /Content/Site.css, для которого есть элемент link в файле /Views/Shared/\_Layout.cshtml. Вы можете увидеть, какие стили мы добавили в Site.css, в листинге 21-3. Мы определим элементы, к которым они применяют, далее в этой главе.

### Листинг 21-3: Добавление стилей в файл Site.css

```
...
table, td, th {
    border: thin solid black; border-collapse: collapse; padding: 5px;
    background-color: lemonchiffon; text-align: left; margin: 10px 0;
}
div.load {color: red; margin: 10px 0; font-weight: bold;}
div.ajaxLink {margin-top: 10px; margin-right: 5px; float: left;}
...

```

## Создание базовых ссылок и URL

Одной из самых основных задач в представлении является создание ссылок или URL, по которым пользователь может перейти в другую часть приложения. В предыдущих главах вы видели большую часть вспомогательных методов, которые можно использовать для создания ссылок и URL. И сейчас мы хотим воспользоваться моментом и напомнить их вам, прежде чем перейти к некоторым из более продвинутых вспомогательных методов. В таблице 21-1 описаны вспомогательные методы HTML и показаны примеры с ними.

#### Совет

*Напомним, что выгода от использования этих вспомогательных методов для создания ссылок и URL заключается в том, что выходные данные выводятся из конфигурации маршрутизации, что обозначает, что изменения в роутах автоматически отражаются на ссылках и URL.*

**Таблица 21-1:** Вспомогательные методы HTML, которые отображают URL

Описание	Пример
URL относительно приложения	Url.Content("~/Content/Site.css"); Выход: /Content/Site.css
Ссылка к именованному действию/контроллеру	Html.ActionLink("My Link", "Index", "Home"); Выход: <a href="/">My Link</a>
URL для действия	Url.Action("GetPeople", "People"); Выход: /People/GetPeople
URL с использованием роутовых данных	Url.RouteUrl(new {controller = "People", action="GetPeople"}); Выход: /People/GetPeople
Ссылка с использованием роутовых данных	Html.RouteLink("My Link", new {controller = "People", action="GetPeople"}); Выход: <a href="/People/GetPeople">My Link</a>
Ссылка к именованному роуту	Html.RouteLink("My Link", "FormRoute", new {controller = "People", action="GetPeople"}); Выход: <a href="/app/forms/People/GetPeople">My Link</a>

Чтобы показать эти вспомогательные методы, мы создали файл представления /People/Index.cshtml, содержание которого вы можете увидеть в листинге 21-4.

#### Листинг 21-4: Содержание файла /People/Index.cshtml

```
@{  
    ViewBag.Title = "Index";  
}  


## Basic Links & URLs



| Helper                                                                                  | Output            |
|-----------------------------------------------------------------------------------------|-------------------|
| Url.Content("~/Content/Site.css")                                                       | /Content/Site.css |
| Html.ActionLink("My Link", "Index", "Home")                                             | My Link           |
| Url.Action("GetPeople", "People")                                                       | /People/GetPeople |
| Url.RouteUrl(new {controller = "People", action="GetPeople"})                           | /People/GetPeople |
| Html.RouteLink("My Link", new {controller = "People", action="GetPeople"})              | My Link           |
| Html.RouteLink("My Link", "FormRoute", new {controller = "People", action="GetPeople"}) | My Link           |


```

Это представление содержит тот же набор вспомогательных вызовов, которые мы перечислили в таблице 21-1, и представляет результаты в HTML таблице, как показано на рисунке 21-1. Мы включили этот пример, потому что он позволяет легко экспериментировать с изменением маршрутизации и сразу видеть результат.

**Рисунок 21-1:** Использование вспомогательных методов для создания ссылок и URL



# Использование в MVC "ненавязчивого" (unobtrusive) Ajax

Ajax (часто упоминается как AJAX) является сокращением для *Asynchronous JavaScript and XML*. Как мы увидим, XML часть является не столь значительной, как это было раньше, но асинхронная часть делает Ajax полезным. Это модель для запроса данных с сервера в фоновом режиме, без перезагрузки веб-страницы.

MVC фреймворк содержит встроенную поддержку для *ненавязчивого* Ajax, что означает, что вы используете вспомогательные методы для определения Ajax функций, вместо того чтобы добавлять блоки кода в представления.

## Совет

Функция ненавязчивого Ajax в MVC основана на JQuery. Если вы знакомы с тем, как JQuery обрабатывает Ajax, тогда вы поймете эту особенность MVC очень быстро.

### Создание синхронного представления в виде формы

Мы собираемся начать этот раздел с создания представление для действия GetPeople в нашем контроллере, то есть мы создали файл /Views/People/GetPeople.cshtml. Вы можете увидеть содержимое этого файла в листинге 21-5.

**Листинг 21-5:** Содержимое файла представления GetPeople.cshtml

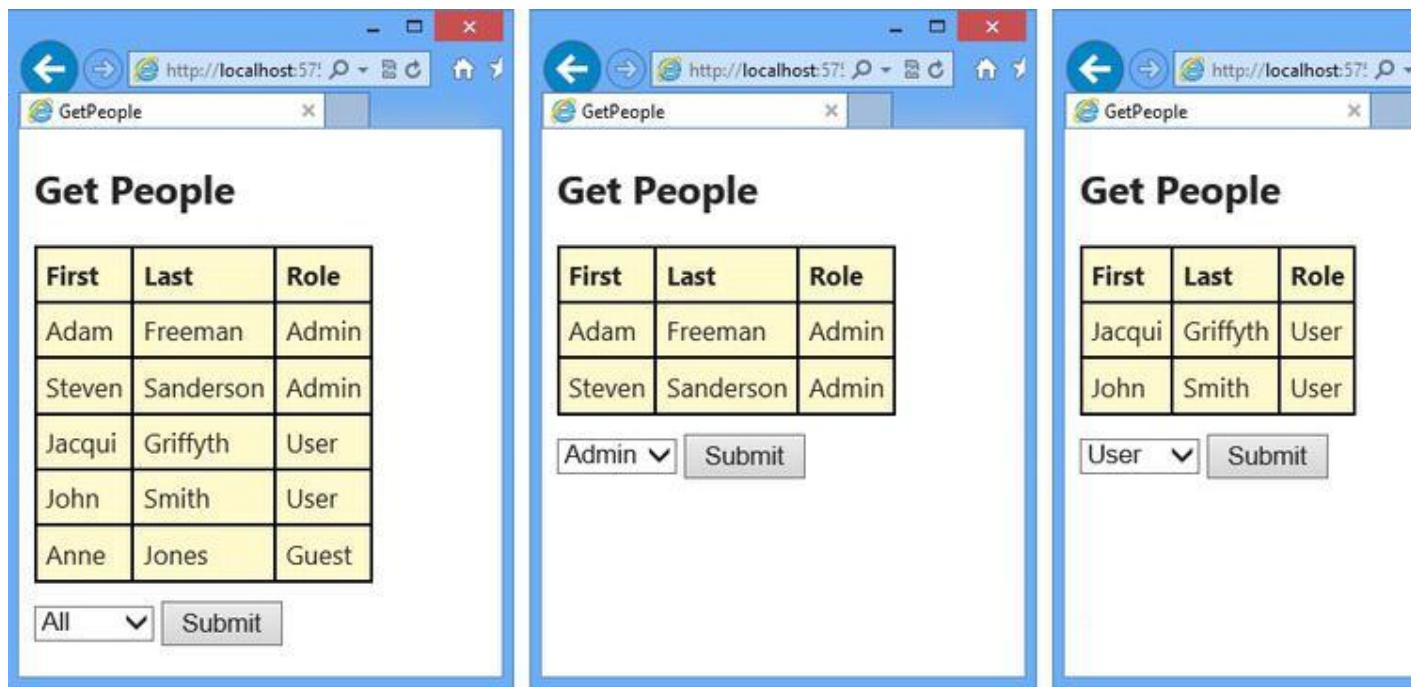
```
@using HelperMethods.Models
@model IEnumerable<Person>
 @{
    ViewBag.Title = "GetPeople";
}
<h2>Get People</h2>
<table>
    <thead><tr><th>First</th><th>Last</th><th>Role</th></tr></thead>
    <tbody>
        @foreach (Person p in Model) {
            <tr>
                <td>@p.FirstName</td>
                <td>@p.LastName</td>
                <td>@p.Role</td>
            </tr>
        }
    </tbody>
</table>
@using (Html.BeginForm()) {
    <div>
        @Html.DropDownList("selectedRole", new SelectList(
            new [] {"All"}.Concat(Enum.GetNames(typeof(Role)))))
        <button type="submit">Submit</button>
    </div>
}
```

Это строго типизированное представление, чьим типом модели является `IEnumerable<Person>`. Мы перечисляем объекты Person в модели для создания строк в HTML таблице и используем вспомогательный метод `Html.BeginForm` для создания простой формы, которая возвращается к действию и контроллеру, сгенерировавшим представление. Форма содержит вызов вспомогательного метода `Html.DropDownList`, который мы используем для создания элемента `select`, содержащего

элементы `option` для каждого из значений, определенных перечислением `Role`, плюс значение `All`. (Мы воспользовались LINQ для создания списка значений для элемента `option` путем объединения значений в `enum` с массивом, который содержит одну строку `All`).

Форма содержит кнопку для отправки формы. Результат заключается в том, что вы можете воспользоваться формой для фильтрации объектов `Person`, которые мы определили в контроллере в листинге 21-2, как показано на рисунке 21-2.

Рисунок 21-2: Простая синхронная форма



Это простая демонстрация фундаментального ограничения в HTML форме, которое заключается в том, что вся страница перезагружается при отправке формы. Это означает, что все содержимое веб-страницы должно быть заново сгенерировано и загружено с сервера (что может быть дорогостоящей операцией для сложных представлений). И в то время, как это происходит, пользователи не могут выполнять любые другие задачи с приложением. Они должны ждать, пока новая страница не сгенерируется, не загрузится, а затем отобразится в браузере.

Для простых приложений, как это, где браузер и сервер работают на одной машине, задержка едва заметна, но для реальных приложений при реальных интернет соединениях синхронные формы могут сделать использование веб приложения неприятным для пользователей и дорогим с точки зрения пропускной способности серверов и вычислительной мощности.

#### Подготовка проекта для ненавязчивого Ajax

Функция ненавязчивого Ajax настроена в двух местах приложения. Во-первых, в файле `/Web.config` (одном из файлов в корневой папке проекта) элемент `configuration/appSettings` содержит запись для свойства `UnobtrusiveJavaScriptEnabled`, которое должно быть установлено на `true`, как показано в листинге 21-6. (Это свойство имеет значение `true` по умолчанию, когда Visual Studio создает проект).

**Листинг 21-6:** Включение ненавязчивого Ajax в файле Web.config

```
...
<configuration>
<!-- другие элементы опущены для краткости -->
<appSettings>
    <add key="webpages:Version" value="2.0.0.0" />
    <add key="webpages:Enabled" value="false" />
    <add key="PreserveLoginUrl" value="true" />
    <add key="ClientValidationEnabled" value="true" />
    <add key="UnobtrusiveJavaScriptEnabled" value="true" />
</appSettings>
<!-- другие элементы опущены для краткости -->
</configuration>
...
```

В дополнение к проверке настойки в Web.config нам нужно добавить ссылки на JavaScript библиотеки jQuery, которые реализуют функциональность ненавязчивого Ajax. Вы можете ссылаться на эти библиотеки из отдельных представлений, но наиболее распространенным подходом является реализация этого в файле макета, поэтому он влияет на все представления, которые используют данный макет. В листинге 21-7, можно увидеть, как мы добавили ссылки на две библиотеки JavaScript в файл /Views/Shared/\_Layout.cshtml.

**Листинг 21-7:** Добавление ссылок для JavaScript библиотек ненавязчивого Ajax в файл \_Layout.cshtml

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    <link href="~/Content/Site.css" rel="stylesheet" />
    <style type="text/css">
        label {
            display: inline-block;
            width: 100px;
        }

        h2 > label {
            width: inherit;
        }

        .dataElem {
            margin: 5px;
        }

        .editor-label, .editor-field {
            float: left;
        }

        .editor-label, .editor-label label, .editor-field input {
            height: 20px;
        }

        .editor-label {
            clear: left;
        }

        .editor-field {
            margin-left: 10px;
            margin-top: 10px;
        }
    </style>
</head>
<body>
```

```

input[type=submit] {
    float: left;
    clear: both;
    margin-top: 10px;
}

.column {
    float: left;
    margin: 10px;
}

```

</style>

<script src="~/Scripts/jquery-1.7.1.min.js" type="text/javascript"></script>

<script src="~/Scripts/jquery.unobtrusive-ajax.min.js" type="text/javascript"></script>

</head>

<body>

    @RenderBody()

</body>

</html>

Файлы, у которых есть ссылки на наши элементы script, добавляются Visual Studio в папку Scripts проекта, когда вы создаете новый MVC проект, используя шаблон Basic. Файл jquery-1.7.1.min.js содержит базовую библиотеку JQuery, а файл jquery.unobtrusive-ajax.min.js содержит функционал Ajax (который опирается на главную библиотеку JQuery). Расширение .min обозначает, что это ужатую версии библиотек, которые меньше, чем версии без расширения .min, но с которыми невозможна отладка. Мы, как правило, не используем .min версии при разработке, а затем переключаемся на них в финальной версии наших проектов.

*Совет*

*JQuery является хорошо поддерживаемой библиотекой, и новые версии появляются часто. Поэтому проверяйте постоянно версии, чтобы знать, что вы используете последнюю. Мы будем использовать версии, которые пришли с первоначальным выпуском Visual Studio 2012, и когда мы писали эту книгу, JQuery 1.8.2 можно было получить на <http://jquery.com>.*

---

## Создание "ненавязчивой" Ajax формы

Теперь мы готовы начать применять ненавязчивого Ajax в нашем приложении, начиная с формы с ненавязчивым Ajax. В последующих разделах мы рассмотрим процесс замены обычных синхронных форм эквивалентным Ajax и объясним, как работает ненавязчивый Ajax.

### Подготовка контроллера

Наша цель состоит в том, чтобы менялись только данные HTML элемента table, когда пользователь нажимает на кнопку Submit в нашем приложении. Это означает, что первое, что нам нужно сделать, это переделать методы действия в нашем контроллере People так, чтобы мы могли получить только те данные, которые нам нужны. Вы можете увидеть изменения, которые мы внесли в контроллер People, в листинге 21-8.

**Листинг 21-8:** Изменение методов действий в контроллере People

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

```

```

using System.Web.Mvc;
using HelperMethods.Models;
namespace HelperMethods.Controllers
{
    public class PeopleController : Controller
    {
        private Person[] personData = {
            new Person {FirstName = "Adam", LastName = "Freeman", Role = Role.Admin},
            new Person {FirstName = "Steven", LastName = "Sanderson", Role = Role.Admin},
            new Person {FirstName = "Jacqui", LastName = "Griffyth", Role = Role.User},
            new Person {FirstName = "John", LastName = "Smith", Role = Role.User},
            new Person {FirstName = "Anne", LastName = "Jones", Role = Role.Guest}
        };
        public ActionResult Index()
        {
            return View();
        }
        public PartialViewResult GetPeopleData(string selectedRole = "All")
        {
            IEnumerable<Person> data = personData;
            if (selectedRole != "All")
            {
                Role selected = (Role)Enum.Parse(typeof(Role), selectedRole);
                data = personData.Where(p => p.Role == selected);
            }
            return PartialView(data);
        }
        public ActionResult GetPeople(string selectedRole = "All")
        {
            return View((object)selectedRole);
        }
    }
}

```

Мы добавили действие `GetPeopleData`, оно выбирает объекты `Person`, которые нам нужно отобразить, и передает их методу `PartialView` для создания требуемых строк таблицы. Поскольку выборка данных обрабатывается методом действия `GetPeopleData`, мы можем сильно упростить метод действия `GetPeople` и полностью удалить версию `HttpPost`. Цель этого метода состоит в передаче выбранной роли в виде `string` в представление.

Мы создали новый файл частичного представления `/Views/People/GetPeopleData.cshtml` для нового метода действия `GetPeopleData`. Вы можете увидеть содержимое представления в листинге 21-9. Это представление отвечает за генерацию элементов `tr`, которые будут заполнять таблицу, используя перечисление объектов `Person`, которые передаются от метода действия.

### Листинг 21-9: Создание представления GetPeopleData.cshtml

```

@using HelperMethods.Models
@model IEnumerable<Person>
@foreach (Person p in Model) {
    <tr>
        <td>@p.FirstName</td>
        <td>@p.LastName</td>
        <td>@p.Role</td>
    </tr>
}

```

Мы также должны обновить представление `/Views/People/GetPeople.cshtml`, которое вы можете увидеть в листинге 21-10.

### Листинг 21-10: Обновление представления GetPeople.cshtml

```

@using HelperMethods.Models
@model string
 @{
    ViewBag.Title = "GetPeople";
}
<h2>Get People</h2>
<table>
    <thead>
        <tr>
            <th>First</th>
            <th>Last</th>
            <th>Role</th>
        </tr>
    </thead>
    <tbody>
        @Html.Action("GetPeopleData", new { selectedRole = Model })
    </tbody>
</table>
@using (Html.BeginForm()) {
<div>
    @Html.DropDownList("selectedRole", new SelectList(
        new [] {"All"}.Concat(Enum.GetNames(typeof(Role)))))
    <button type="submit">Submit</button>
</div>
}

```

Мы изменили тип модели представления на `string`, который мы передаем вспомогательному методу `Html.Action` для вызова дочернего действия `GetPeopleData`. Это рендерит частичное представление и дает нам строки таблицы.

## Создание Ajax формы

У нас после этих изменений еще есть синхронная форма в приложении, но мы выделили функциональность контроллера таким образом, мы можем запрашивать только строки таблицы через действие `GetPeopleData`. Этот новый метод действия будет целью нашего Ajax запроса, и следующим шагом является обновление представления `GetPeople.cshtml`, таким образом, чтобы работа с формой осуществлялась через Ajax, как показано в листинге 21-11.

**Листинг 21-11:** Создание ненавязчивой Ajax формы в представлении `GetPeople.cshtml`

```

@using HelperMethods.Models
@model string
 @{
    ViewBag.Title = "GetPeople";
    AjaxOptions ajaxOpts = new AjaxOptions {
        UpdateTargetId = "tableBody"
    };
}
<h2>Get People</h2>
<table>
    <thead>
        <tr>
            <th>First</th>
            <th>Last</th>
            <th>Role</th>
        </tr>
    </thead>
    <tbody id="tableBody">
        @Html.Action("GetPeopleData", new { selectedRole = Model })
    </tbody>
</table>
@using (Ajax.BeginForm("GetPeopleData", ajaxOpts)) {

```

```

<div>
    @Html.DropDownList("selectedRole", new SelectList(
        new [] {"All"}).Concat(Enum.GetNames(typeof(Role))))))
    <button type="submit">Submit</button>
</div>
}

```

Как вы можете видеть, изменения относительно мелкие, и нам не нужно вставлять блоки кода вокруг представления, чтобы все работало.

В центре поддержки MVC фреймворком Ajax форм находится вспомогательный метод `Ajax.BeginForm`, который принимает в качестве аргумента объект `AjaxOptions`. Нам нравится создавать объекты `AjaxOptions` при запуске представления в блоке кода Razor, но вы можете создать их внутри, когда вызываете `Ajax.BeginForm`.

Класс `AjaxOptions`, находящийся в пространстве имен `System.Web.Mvc.Ajax`, определяет свойства, которые позволяют нам конфигурировать то, как делается асинхронный запрос к серверу и что происходит с данными, которые мы получаем обратно. Эти свойства описаны в таблице 21-2.

**Таблица 21-2:** Свойства `AjaxOptions`

Свойство	Описание
Confirm	Задает сообщение, которое будет отображаться для пользователя в окне подтверждения, прежде чем будет сделан Ajax запрос.
HttpMethod	Задает HTTP метод, который будет использоваться для создания запроса. Это может быть либо <code>Get</code> , либо <code>Post</code> .
InsertionMode	Указывает способ, которым содержание, полученное с сервера, будет вставлено в HTML. Есть три варианта выбора: <code>InsertAfter</code> , <code>InsertBefore</code> и <code>Replace</code> (по умолчанию).
LoadingElementId	Указывает ID HTML элемента, который будет отображаться, пока выполняется Ajax запрос.
LoadingElementDuration	Задает продолжительность анимации, которая используется для выявления элемента, определенного <code>LoadingElementId</code> .
UpdateTargetId	Указывает ID HTML элемента, в который будет вставлен контент, полученный от сервера.
Url	Задает Url, который будет запрашиваться сервером.

В листинге мы установили свойство `UpdateTargetId` для `tableBody`. Это `id`, которое мы присвоили HTML элементу `tbody` в представлении в листинге 21-11. Когда пользователь нажмет на кнопку `Submit`, асинхронный запрос будет сделан к методу действия `GetPeopleData`, а возвращенный HTML фрагмент будет использоваться для замены существующих элементов в `tbody`.

#### *Совет*

*Класс `AjaxOptions` также определяет свойства, которые позволяют нам указать функции обратного вызова для различных этапов жизненного цикла запроса, см. «Работа с функциями обратного вызова Ajax» далее в этой главе.*

Ну вот и все: мы заменим метод `Html.BeginForm` `Ajax.BeginForm` и убеждаемся, что у нас есть цель для нового содержания. Все остальное происходит автоматически, и мы получаем асинхронную форму.

## Как работает ненавязчивый Ajax

Когда мы вызываем вспомогательный метод `Ajax.BeginForm`, параметры, которые мы указали при помощи объекта `AjaxOptions`, превращаются в атрибуты, применяемые к элементу `form`. Наше представление в листинге 21-11 создает следующий элемент `form`:

```
...
<form action="/People/GetPeopleData" data-ajax="true" data-ajax-mode="replace"
      data-ajax-update="#tableBody" id="form0" method="post">
...

```

Когда HTML страница, полученная из представления `GetPeople.cshtml`, загружается в браузер, JavaScript в библиотеке `jquery.unobtrusive-ajax.js` сканирует HTML элементы и определяет Ajax форму, когда ищет элементы, у которых атрибуты `data- ajax` имеют значение `true`.

Остальные атрибуты, имена которых начинаются с `data- ajax`, содержат значения, которые мы задали с помощью класса `AjaxOptions`. Эти параметры конфигурации используются для настройки JQuery, который имеет встроенную поддержку для управления Ajax запросами.

### *Совет*

*Вы не обязательно должны использовать MVC фреймворк для поддержки ненавязчиво Ajax. Есть много альтернатив, в том числе использование JQuery напрямую. То есть, вы можете выбрать определенную технику и придерживаться ее. Мы не рекомендуем смешивать MVC поддержку ненавязчивого Ajax с другими методами и библиотеками в одном и том же представлении, поскольку это может вызвать некоторые проблемы, такие как дублирование или падение Ajax запросов.*

## Установка опций Ajax

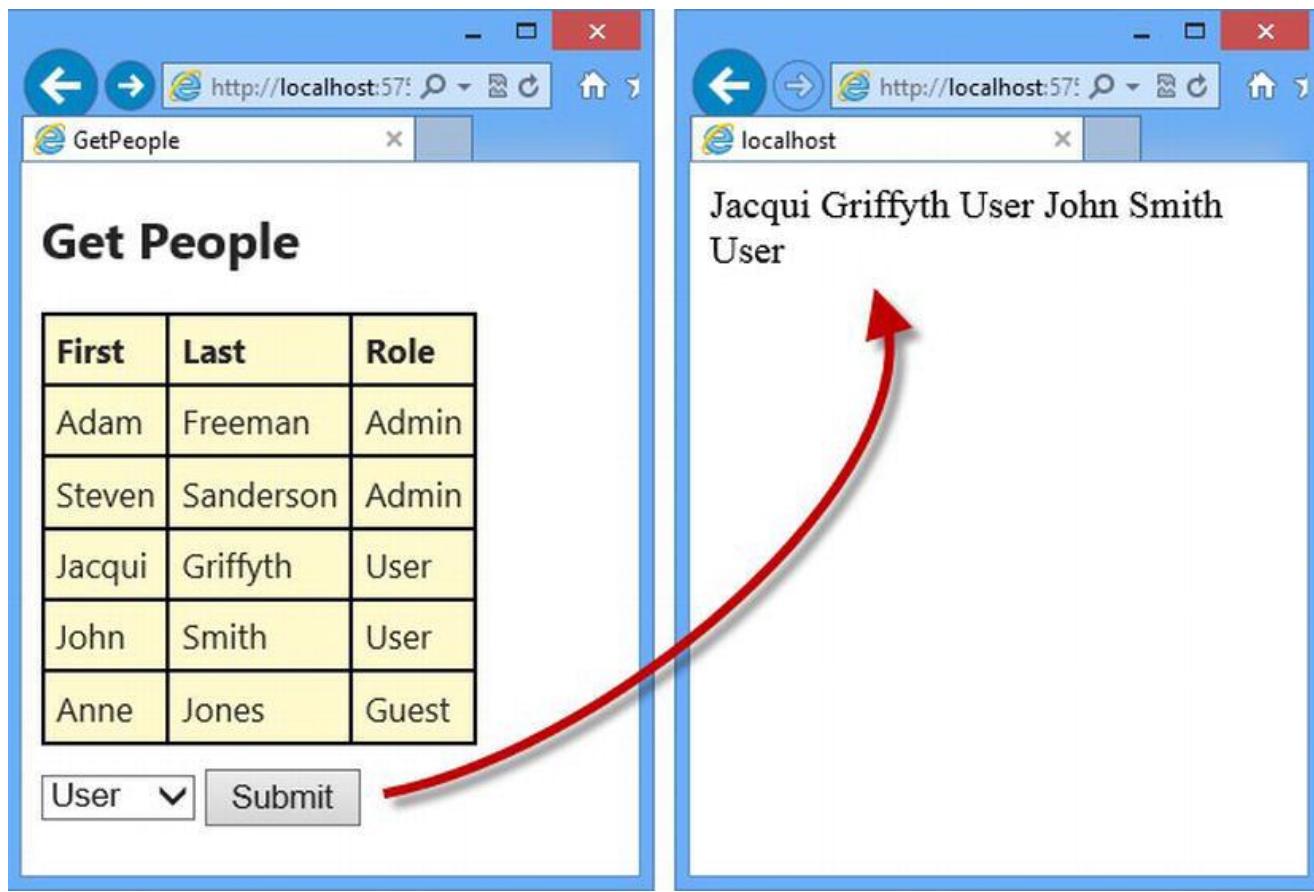
Мы можем точно настроить поведение наших Ajax запросов, установив значения для свойств объекта `AjaxOptions`, который мы передаем вспомогательному методу `Ajax.BeginForm`. В следующих разделах мы объясняем, что каждая из этих опций делает и почему они могут быть полезны.

### Обеспечение постепенного ухудшения

Когда мы создали форму с поддержкой Ajax в листинге 21-11, мы асинхронно передали имя метода действия, который должен быть вызван. В нашем примере это было действие `GetPeopleData`, которое генерирует частичное представление, содержащее фрагмент HTML.

Одна проблема с этим подходом заключается в том, что все это не очень хорошо работает, если пользователь отключил JavaScript (или использует браузер, который не поддерживает его). В таких случаях, когда пользователь отправляет форму, браузер отменяет отображение текущий HTML страницы и заменяет ее фрагментом, возвращенным целевым методом действия. Результат показан на рисунке 21-3.

**Рисунок 21-3:** Результат использования вспомогательного метода Ajax.BeginForm без браузерной поддержки JavaScript



Самым простым способом решения этой проблемы является использование свойства AjaxOptions.Url для определения целевого URL для асинхронного запроса вместо указания имени действия в качестве аргумента метода Ajax.BeginForm, как показано в листинге 21-12.

**Листинг 21-12:** Использование свойства AjaxOptions.Url для обеспечения постепенного ухудшения

```
@using HelperMethods.Models
@model string
 @{
    ViewBag.Title = "GetPeople";
    AjaxOptions ajaxOpts = new AjaxOptions {
        UpdateTargetId = "tableBody",
        Url = Url.Action("GetPeopleData")
    };
}
<h2>Get People</h2>
<table>
    <thead>
        <tr>
            <th>First</th>
            <th>Last</th>
            <th>Role</th>
        </tr>
    </thead>
    <tbody id="tableBody">
        @Html.Action("GetPeopleData", new { selectedRole = Model })
    </tbody>
</table>
```

```

@using (Ajax.BeginForm.ajaxOpts)) {


@Html.DropDownList("selectedRole", new SelectList(
        new [] {"All"}.Concat(Enum.GetNames(typeof(Role)))))
    <button type="submit">Submit</button>


}

```

Мы использовали вспомогательный метод `Url.Action` для создания URL, который будет вызывать действие `GetPeopleData`, и использовали версию метода `Ajax.BeginForm`, который принимает только параметр `AjaxOptions`. В результате этого создается элемент `form`, который отправляется обратно в исходный метод действия, если JavaScript не включен:

```

...
<form action="/People/GetPeople" data-ajax="true" data-ajax-mode="replace"
    data-ajax-update="#tableBody" data-ajax-url="/People/GetPeopleData" id="form0"
    method="post">
...

```

Если JavaScript включен, то библиотека ненавязчивого Ajax примет целевой URL из атрибута `data-ajax-url`, который относится к дочернему действию. Если JavaScript отключен, то браузер будет использовать обычную методику размещения формы, когда целевой URL принимается от атрибута `action`, указывающего обратно на метод действия, который будет генерировать полную HTML страницу.

#### **Внимание**

*Вы можете удивиться, почему мы так заботимся о пользователях, которые отключили JavaScript. И кто же эти пользователи? На самом деле, такие пользователи делятся на две группы. Первая группа состоит из тех, кто очень серьезно воспринимает свою IT-безопасность и отключает все, что может быть использовано для атак: а этим JavaScript славится уже на протяжении многих лет. Вторая группа – это пользователи в крупных корпорациях, где применяется невероятная ограничительная политика во имя IT-безопасности (хотя, по нашему опыту, корпоративные ПК так плохо настроены, что безопасности тут просто не существует, а ограничения лишь раздражают пользователей). Вы можете игнорировать постепенное ухудшение, если чувствуете, что можете игнорировать экспертов по безопасности и людей, которые работают для крупных компаний. Но так как они могут быть технически подкованными и знающими пользователями, мы всегда уделяем время на то, чтобы не оставлять их в стороне.*

---

## **Предоставление пользователям обратной связи при создании Ajax запроса**

Один из недостатков использования Ajax заключается в том, что для пользователя не очевидно, что что-то происходит, потому что запрос к серверу производится в фоновом режиме. Мы можем сообщить пользователю, что запрос выполняется, при помощи свойств `AjaxOptions.LoadingElementId` и `AjaxOptions.LoadingElementDuration`. В листинге 21-13 показано, как мы применили эти свойства для файла представления `GetPerson.cshtml`.

**Листинг 21-13:** Создание обратной связи с пользователем при помощи свойства `LoadingElementId`

```

@using HelperMethods.Models
@model string
 @{

```

```

ViewBag.Title = "GetPeople";
AjaxOptions ajaxOpts = new AjaxOptions {
    UpdateTargetId = "tableBody",
    Url = Url.Action("GetPeopleData"),
    LoadingElementId = "loading",
    LoadingElementDuration = 1000,
};
}
<h2>Get People</h2>
<div id="loading" class="load" style="display: none">
    <p>Loading Data...</p>
</div>
<table>
    <thead>
        <tr>
            <th>First</th>
            <th>Last</th>
            <th>Role</th>
        </tr>
    </thead>
    <tbody id="tableBody">
        @Html.Action("GetPeopleData", new { selectedRole = Model })
    </tbody>
</table>
@using (Ajax.BeginForm(ajaxOpts)) {
<div>
    @Html.DropDownList("selectedRole", new SelectList(
    new [] {"All"}.Concat(Enum.GetNames(typeof(Role)))))
    <button type="submit">Submit</button>
</div>
}

```

Свойство AjaxOptions.LoadingElementId определяет значение атрибута id скрытого HTML элемента, который будет показан пользователю, когда выполняется Ajax запрос. Чтобы показать эту возможность, мы добавили к представлению элемент div, который мы скрыли от пользователя, установив значение CSS свойства display на none. Мы дали этому элементу div id атрибут loading и использовали этот id в качестве значения для свойства LoadingElementId. Ненавязчивый Ajax будет отображать элемент для пользователя во время выполнения запроса, как показано на рисунке 21-4. Свойство LoadingElementDuration задает продолжительность анимации, который используется, чтобы показать элемент loading пользователю. Мы определили значение 1000, которое обозначает одну секунду.

**Рисунок 21-4:** Обратная связь с пользователем во время выполнения Ajax запроса



## Подсказка для пользователя перед тем, как он сделал запрос

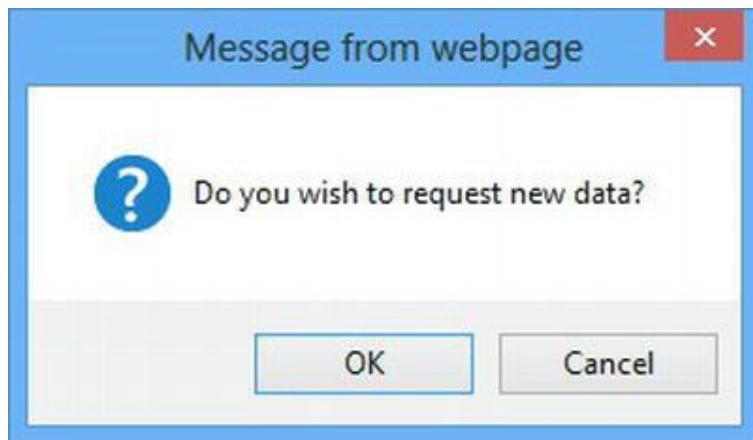
Свойство AjaxOptions.Confirm позволяет нам указать сообщение, которое будет использоваться как подсказка для пользователя перед каждым асинхронным запросом. Пользователь может выбрать дальнейшее выполнение или отмену запроса. В листинге 21-14 показано, как мы применили это свойство в файле GetPerson.cshtml.

**Листинг 21-14:** Подсказка для пользователя перед тем, как он сделал запрос

```
...
@{
    ViewBag.Title = "GetPeople";
    AjaxOptions ajaxOpts = new AjaxOptions {
        UpdateTargetId = "tableBody",
        Url = Url.Action("GetPeopleData"),
        LoadingElementId = "loading",
        LoadingElementDuration = 1000,
        Confirm = "Do you wish to request new data?"
    };
}
...
...
```

С этим дополнением пользователю каждый раз при отправки формы показывается окно с сообщением, как представлено на рисунке 21-5. Это окно предлагается пользователю при каждом запросе. Это означает, что данную функцию следует использовать с осторожностью, чтобы не раздражать пользователя.

**Рисунок 21-5:** Окно с сообщением для пользователя перед отправкой запроса



## Создание Ajax ссылок

В дополнение к формам мы можем использовать ненавязчивый Ajax для создания асинхронных элементов a. Механизм этого очень похоже на то, как работают Ajax формы. Вы можете увидеть, как мы добавили ссылки в представление Ajax GetPeople.cshtml, в листинге 21-15.

**Листинг 21-15:** Создание ссылок при помощи Ajax

```
@using HelperMethods.Models
@model string
 @{
    ViewBag.Title = "GetPeople";
    AjaxOptions ajaxOpts = new AjaxOptions {
        UpdateTargetId = "tableBody",
        Url = Url.Action("GetPeopleData"),
        ...
    };
}
```

```

        LoadingElementId = "loading",
        LoadingElementDuration = 1000,
        Confirm = "Do you wish to request new data?"
    };
}
<h2>Get People</h2>
<div id="loading" class="load" style="display: none">
    <p>Loading Data...</p>
</div>
<table>
    <thead>
        <tr>
            <th>First</th>
            <th>Last</th>
            <th>Role</th>
        </tr>
    </thead>
    <tbody id="tableBody">
        @Html.Action("GetPeopleData", new { selectedRole = Model })
    </tbody>
</table>
@using (Ajax.BeginForm.ajaxOpts)) {
<div>
    @Html.DropDownList("selectedRole", new SelectList(
        new [] {"All"}.Concat(Enum.GetNames(typeof(Role)))))
    <button type="submit">Submit</button>
</div>
}
<div>
    @foreach (string role in Enum.GetNames(typeof(Role))) {
        <div class="ajaxLink">
            @Ajax.ActionLink(role, "GetPeopleData",
                new { selectedRole = role },
                new AjaxOptions { UpdateTargetId = "tableBody" })
        </div>
    }
</div>

```

Мы использовали цикл `foreach`, чтобы вызвать вспомогательный метод `Ajax.ActionLink` для каждого из значений, определенных перечислением `Role`, создав набор элементов `a` с поддержкой Ajax. Созданные элементы `a` имеют такие же атрибуты `data`, которые мы видели при работе с формами:

```

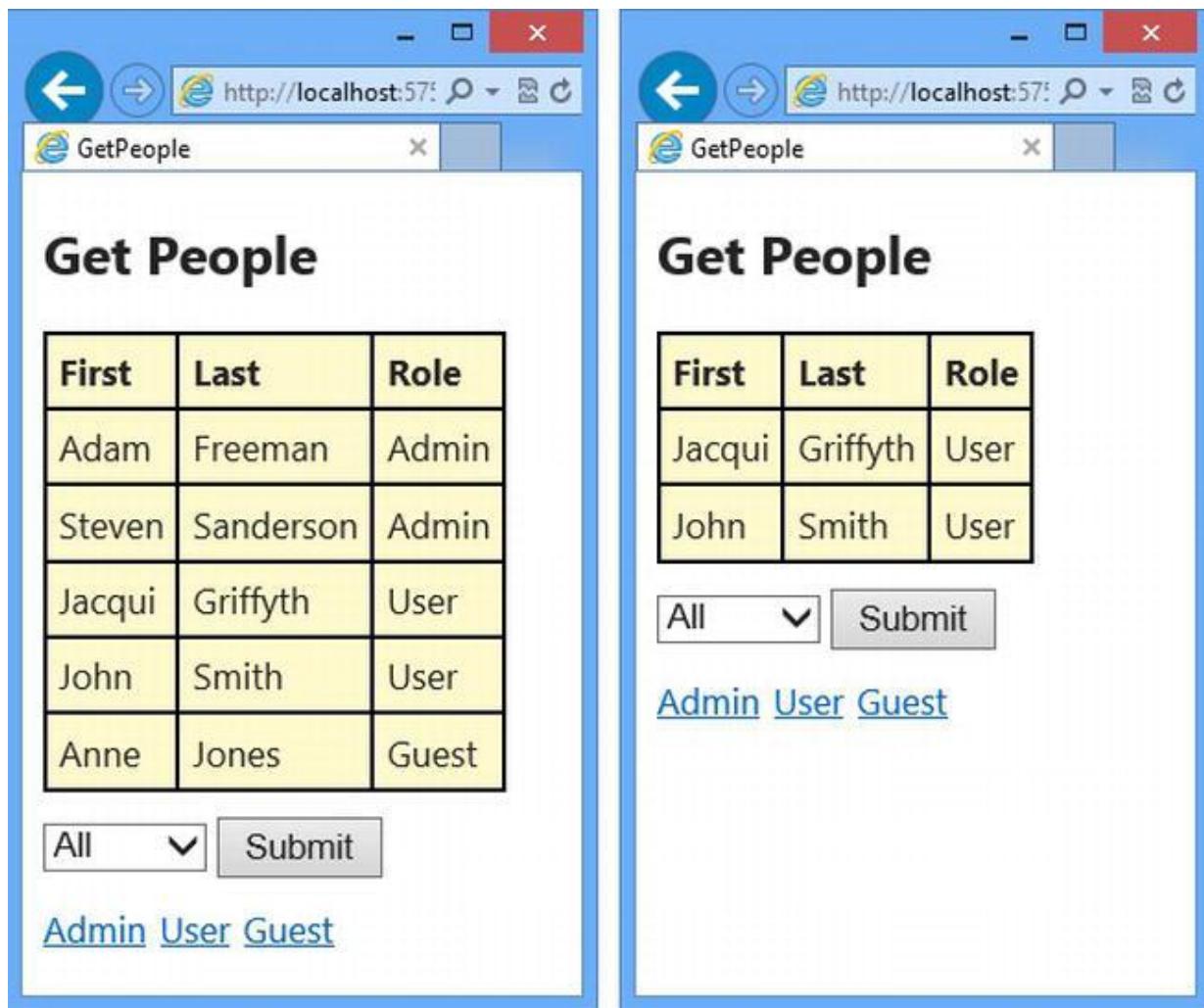
...
<a data-ajax="true" data-ajax-mode="replace" data-ajax-update="#tableBody"
    href="/People/GetPeopleData?selectedRole=Guest">Guest</a>
...

```

Наша конфигурация маршрутизации не имеет записи для переменной `selectedRole`, поэтому URL, который был создан для атрибута `href`, определяет ту роль, которую представляет ссылка, используя строковый компонент запроса URL.

Вы можете увидеть ссылки, которые мы добавили в представление, на рисунке 21-6. Нажатие по одной из этих ссылок вызовет метод действия `GetPersonData` действия и заменит содержимое элемента `tbody` возвращенным HTML фрагментом. Это создает тот же результат фильтрации данных, которого мы достигли с помощью Ajax формы ранее в этой главе.

Рисунок 21-6: Использование Ajax ссылок в представлении



*Совет*

*Возможно, вам понадобиться очистить историю в браузере, чтобы увидеть изменения, представленные в данном примере.*

## Обеспечение постепенного ухудшения для ссылок

Мы столкнулись с той же проблемой с Ajax ссылками, которая у нас возникла с Ajax формами: когда нет браузерной поддержки JavaScript, нажатие по одной из ссылок будет просто отображать HTML фрагмент, который генерирует метод действия GetPeopleData.

Мы решаем эту проблему с помощью свойства `AjaxOptions.Url`, чтобы указать URL для Ajax запроса, и мы задаем действие GetPeople для вспомогательного метода `Ajax.ActionLink`, как показано в листинге 21-16.

### Листинг 21-16: Создание постепенного ухудшения для Ajax ссылок

```
<div>
    @foreach (string role in Enum.GetNames(typeof(Role))) {
        <div class="ajaxLink">
            @Ajax.ActionLink(role, "GetPeople",
                new { selectedRole = role },
```

```

        new AjaxOptions {
            UpdateTargetId = "tableBody",
            Url = Url.Action("GetPeopleData", new { selectedRole = role })
        })
    </div>
}
</div>
...

```

Именно поэтому мы создаем новый объект AjaxOptions для каждой из требуемых ссылок, а не использовали ту, которую мы создали в блоке кода Razor для элемента form. Независимый AjaxOptions позволяет нам указать различные значения для свойства Url для каждой ссылки и поддерживает постепенное ухудшение для браузеров без JavaScript.

## Работа с функциями обратного вызова Ajax

Класс AjaxOptions определяет набор свойств, которые позволяют нам указать JavaScript функции, которые будут вызываться в различных точках жизненного цикла Ajax запроса. Эти свойства описаны в таблице 21-3.

**Таблица 21-3:** Свойства обратного вызова AjaxOptions

Свойство	Событие jQuery	Описание
OnBegin	beforeSend	Вызывается непосредственно перед отправлением запроса
OnComplete	complete	Вызывается, если запрос был выполнен, независимо от того, успешно или нет
OnFailure	error	Вызывается, если запрос выполнен неудачно
OnSuccess	success	Вызывается, если запрос выполнен успешно

Каждое из свойств обратного вызова AjaxOptions соотносится с событием Ajax, поддерживаемым библиотекой JQuery. Мы перечислили JQuery события в таблице 21-3 для тех читателей, которые раньше использовали JQuery. Вы можете получить подробную информацию о каждом из этих событий и параметрах, которые будут переданы вашей функции, на <http://api.jquery.com/jQuery.ajax>.

В листинге 21-17 можно увидеть, как мы использовали элемент script для определения некоторых базовых функций JavaScript, которые будут отчитываться о ходе Ajax запроса, и использовали свойства, показанные в таблице 21-3, чтобы указать наши функции в качестве обработчиков Ajax событий.

**Листинг 21-17:** Использование функций обратного вызова Ajax

```

@using HelperMethods.Models
@model string
{
    ViewBag.Title = "GetPeople";
    AjaxOptions ajaxOpts = new AjaxOptions {
        UpdateTargetId = "tableBody",
        Url = Url.Action("GetPeopleData"),
        LoadingElementId = "loading",
        LoadingElementDuration = 1000,
        Confirm = "Do you wish to request new data?"
    };
}
<script type="text/javascript">
    function OnBegin() {

```

```

        alert("This is the OnBegin Callback");
    }
    function OnSuccess(data) {
        alert("This is the OnSuccessCallback: " + data);
    }
    function onFailure(request, error) {
        alert("This is the onFailure Callback: " + error);
    }
    function onComplete(request, status) {
        alert("This is the onComplete Callback: " + status);
    }

```

</script>

<h2>Get People</h2>

<div id="loading" class="load" style="display: none">

<p>Loading Data...</p>

</div>

<table>

<thead>

<tr>

<th>First</th>

<th>Last</th>

<th>Role</th>

</tr>

</thead>

<tbody id="tableBody">

@Html.Action("GetPeopleData", new { selectedRole = Model })

</tbody>

</table>

@using (Ajax.BeginForm.ajaxOpts) {

<div>

@Html.DropDownList("selectedRole", new SelectList(

new [] {"All"}.Concat(Enum.GetNames(typeof(Role)))))

<button type="submit">Submit</button>

</div>

}

<div>

@foreach (string role in Enum.GetNames(typeof(Role))) {

<div class="ajaxLink">

@Ajax.ActionLink(role, "GetPeople",

new { selectedRole = role},

new AjaxOptions {

UpdateTargetId = "tableBody",

Url = Url.Action("GetPeopleData", new { selectedRole = role}),

OnBegin = "OnBegin",

OnFailure = "OnFailure",

OnSuccess = "OnSuccess",

OnComplete = "OnComplete"

})

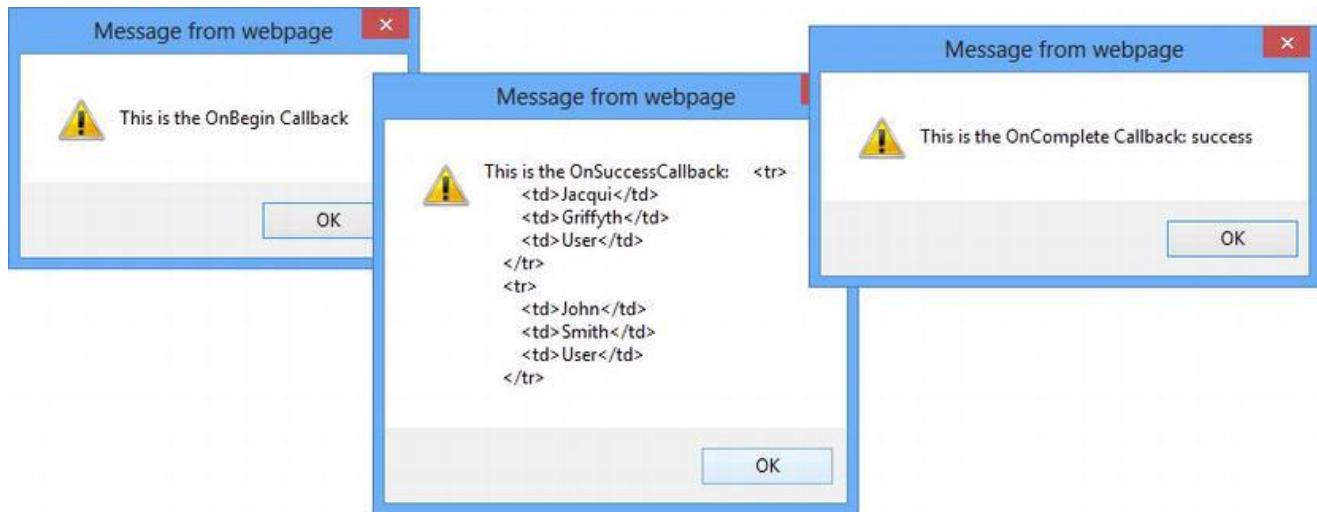
</div>

}

</div>

Мы определили четыре функции, по одному для каждого из обратных вызовов. Это очень легкий пример, и мы просто выводим сообщение для пользователя для каждой из этих функций. С учетом этих изменений переход по одной из ссылок будет отображать последовательность окон, которые сообщают о ходе Ajax запроса, как показано на рисунке 21-7.

**Рисунок 21-7:** Диалоговые окна, которые отображаются в ответ на функцию обратного вызова Ajax



Отображение диалоговых окон пользователю при каждом обратном вызове – не самая полезная вещь, которую можно сделать с обратными вызовами Ajax, но она демонстрирует последовательность, в которой вызываются функции. Мы можем сделать все, что угодно с этими функциями JavaScript: управлять HTML DOM, создавать дополнительные запросы и так далее. Одна из самых полезных вещей, которую мы можем сделать обратным вызовом, это обработка данных JSON. Мы опишем их в следующем разделе.

## Работа с JSON

До сих пор в наших Ajax примерах сервер обрабатывал HTML фрагменты и отправлял их браузеру. Это вполне приемлемая техника, но она громоздка (потому что мы посылаем HTML элементы вместе с данными), и она ограничивает то, что мы можем делать с данными в браузере.

Одним из способов решения обеих этих проблем является использование формата JSON (*JavaScript Object Notation*), который является независимым от языка способом выражения данных. Он возник из языка JavaScript, но давно уже начал свою собственную жизнь и очень широко используется. В этом разделе мы покажем вам, как создать метод действия, который возвращает JSON данные, а также как обрабатывать эти данные в браузере.

### *Совет*

*В главе 25 мы описываем Web API, который является альтернативным подходом для создания веб-сервисов.*

### Добавление поддержки JSON в контроллер

MVC фреймворк делает создание метода действия, который генерирует JSON данные, а не HTML, очень простым. В листинге 21-18 показано, как мы добавили такой метод действия в контроллер People.

#### **Листинг 21-18:** Метод действия, который генерирует данные JSON

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
```

```

using System.Web.Mvc;
using HelperMethods.Models;
namespace HelperMethods.Controllers
{
    public class PeopleController : Controller
    {
        private Person[] personData = {
            new Person {FirstName = "Adam", LastName = "Freeman", Role = Role.Admin},
            new Person {FirstName = "Steven", LastName = "Sanderson", Role = Role.Admin},
            new Person {FirstName = "Jacqui", LastName = "Griffyth", Role = Role.User},
            new Person {FirstName = "John", LastName = "Smith", Role = Role.User},
            new Person {FirstName = "Anne", LastName = "Jones", Role = Role.Guest}
        };
        public ActionResult Index()
        {
            return View();
        }
        private IEnumerable<Person> GetData(string selectedRole)
        {
            IEnumerable<Person> data = personData;
            if (selectedRole != "All")
            {
                Role selected = (Role)Enum.Parse(typeof(Role), selectedRole);
                data = personData.Where(p => p.Role == selected);
            }
            return data;
        }
        public JsonResult GetPeopleDataJson(string selectedRole = "All")
        {
            IEnumerable<Person> data = GetData(selectedRole);
            return Json(data, JsonRequestBehavior.AllowGet);
        }
        public PartialViewResult GetPeopleData(string selectedRole = "All")
        {
            return PartialView(GetData(selectedRole));
        }
        public ActionResult GetPeople(string selectedRole = "All")
        {
            return View((object)selectedRole);
        }
    }
}

```

Поскольку мы хотим представить одни и те же данные в двух различных форматах (HTML и JSON), мы переделали наш контроллер так, чтобы был общий (и private) метод `GetData`, который несет ответственность за выполнение фильтрации.

Мы добавили новый метод действий `GetPeopleDataJson`, который возвращает объект `JsonResult`. Это особый вид `ActionResult`, который говорит движку представления, что мы хотим вернуть клиенту JSON данные, а не HTML. (Вы можете узнать больше о классе `ActionResult` и роли, которую он играет в MVC фреймворке, в главе 15).

Мы создаем `JsonResult` при помощи метода `Json` в методе действия, передавая данные, которые мы хотим преобразовать в формат JSON:

```

...
return Json(data, JsonRequestBehavior.AllowGet);
...

```

В данном случае мы также передали в `AllowGet` значение из перечисления `JsonRequestBehavior`. По умолчанию данные JSON будут отправлены только в ответ на запрос POST, но, передавая это

значение в качестве параметра методу `Json`, мы говорим MVC фреймворку также реагировать на запросы GET.

#### **Внимание**

*Вы должны использовать только `JsonRequestBehavior.AllowGet`, если данные, которые вы возвращаете, не являются закрытыми. В связи с проблемой безопасности во многих веб браузерах для сторонних сайтов становится возможным перехватывать данные JSON, которые вы возвращаете в ответ на запрос GET, поэтому `JsonResult` не будет отвечать на запросы GET по умолчанию. Вместо этого в большинстве случаев вы сможете использовать POST запросы для получения JSON данных, избегая проблемы. Для дополнительной информации см. <http://haacked.com/archive/2009/06/25/json-hijacking.aspx>.*

## Обработка JSON в браузере

Для обработки JSON мы возвращаемся к серверу приложений MVC фреймворка, мы определяем функцию JavaScript, используя свойство обратного вызова `OnSuccess` в классе `AjaxOptions`. В листинге 21-19 вы можете увидеть, как мы обновили файл представления `GetPerson.cshtml`, чтобы удалить функции обработчика, которые мы определили в предыдущем разделе, и использовали `OnSuccess` для обработки данных JSON.

**Листинг 21-19:** Работа с данными JSON в представлении GetPerson

```
@using HelperMethods.Models
@model string
 @{
    ViewBag.Title = "GetPeople";
    AjaxOptions ajaxOpts = new AjaxOptions {
        UpdateTargetId = "tableBody",
        Url = Url.Action("GetPeopleData"),
        LoadingElementId = "loading",
        LoadingElementDuration = 1000,
        Confirm = "Do you wish to request new data?"
    };
}
<script type="text/javascript">
    function processData(data) {
        var target = $("#tableBody");
        target.empty();
        for (var i = 0; i < data.length; i++) {
            var person = data[i];
            target.append("<tr><td>" + person.FirstName + "</td><td>" +
                person.LastName + "</td><td>" + person.Role + "</td></tr>");
        }
    }
</script>
<h2>Get People</h2>
<div id="loading" class="load" style="display: none">
    <p>Loading Data...</p>
</div>
<table>
    <thead>
        <tr>
            <th>First</th>
            <th>Last</th>
            <th>Role</th>
        </tr>
    </thead>
```

```

<tbody id="tableBody">
    @Html.Action("GetPeopleData", new { selectedRole = Model })
</tbody>
</table>
@using (Ajax.BeginForm.ajaxOpts)) {
<div>
    @Html.DropDownList("selectedRole", new SelectList(
        new [] {"All"}.Concat(Enum.GetNames(typeof(Role)))))
    <button type="submit">Submit</button>
</div>
}
<div>
    @foreach (string role in Enum.GetNames(typeof(Role))) {
        <div class="ajaxLink">
            @Ajax.ActionLink(role, "GetPeople",
                new { selectedRole = role },
                new AjaxOptions {
                    Url = Url.Action("GetPeopleDataJson", new { selectedRole = role }),
                    OnSuccess = "processData"
                })
        </div>
    }
</div>

```

Мы определили новую функцию `ProcessData`, содержащую немного базового JQuery кода, который обрабатывает JSON объекты и использует их для создания элементов `tr` и `td`, необходимых для заполнения таблицы.

## Совет

Мы не углубляемся в JQuery в данной книге, потому что он сам по себе является обширной темой. Хотя мы любим JQuery, и если вы хотите узнать о нем больше, то обратите внимание на книгу Адама Фримана [Pro JQuery](#) (Apress, 2012).

Обратите внимание, что мы удалили значение для свойства `UpdateTargetId` из объектов `AjaxOptions`, которые мы создали для ссылок. Если вы забыли это сделать, ненавязчивый Ajax постарается обрабатывать данные JSON, полученные от сервера, как HTML. Обычно это происходит потому, что содержание целевого элемента удаляется, но не заменяется новыми данными.

Вы можете увидеть результат перехода к JSON, запустив приложение, перейдя по URL `/People/GetPerson` и нажав на одну из ссылок. Как показано на рисунке 21-8, мы получаем не совсем правильный результат: в частности, информация, отображаемая в колонке `Role` таблицы, не является правильной. Мы объясним, почему это происходит, и покажем вам, как это исправить, в следующем разделе.

**Рисунок 21-8:** Работа с данными JSON вместо HTML



## Подготовка данных для кодирования

Когда мы вызвали метод `Json` внутри метода действия `GetPeopleDataJson`, мы оставили MVC фреймворк выяснить, как кодировать наши объекты `Person` в формате JSON. MVC фреймворк не имеет специального знания о нашей модели, и поэтому он очень старается предположить, что он должен делать. Вот как MVC фреймворк выражает один объект `Person` в JSON:

```
{"PersonId":0,"FirstName":"Adam","LastName":"Freeman",
"BirthDate": "\/Date(6213559680000) \/", "HomeAddress":null,"IsApproved":false,"Role":0}
...
```

Это похоже на кашу, но результат на самом деле довольно разумный: это просто не совсем то, что нам надо. Во-первых, все свойства, определенные классом `Person`, представлены в JSON, хотя мы и не присваивали им значения в контроллере `People`. В некоторых случаях используется значение по умолчанию для данного типа (например, `false` используется для `IsApproved`), а в других случаях используется `null` (например, для `HomeAddress`). Некоторые значения преобразуются так, чтобы их мог легко истолковать JavaScript, такие как свойство `BirthDate`, а другие обрабатываются не так хорошо, например, использование `0` для свойства `Role`, а не `Admin`.

## Просмотр данных JSON

Полезно посмотреть, какие JSON данные возвращают методы действий, и самый простой способ сделать это – ввести URL, направленный на действие, в браузер:

```
http://localhost:57520/People/GetPeopleJson?selectedRole=Admin
```

Вы можете сделать это фактически в любом браузере, но большинство браузеров заставят вас сохранить и открыть текстовый файл, прежде чем вы сможете увидеть содержимое JSON. Нам нравится использовать для этого браузер Google Chrome, потому что он показывает JSON данные в главном окне браузера, что делает процесс быстрее, и обозначает, что в итоге вы не будете сидеть с десятками открытых окон текстового файла. Мы также рекомендуем Fiddler ([www.fiddler2.com](http://www.fiddler2.com)), который является отличной прокси веб отладки и позволяет просмотреть всю информацию о данных, передаваемых между браузером и сервером.

MVC фреймворк сделал хорошую попытку, но мы в конечном итоге отправили в браузер свойства, которые мы в дальнейшем не будем использовать, а значение `Role` не является полезным для нас. Это типичная ситуация, когда перед отправкой данных клиенту требуется немного подготовиться.

В листинге 21-20 можно увидеть, как мы переделали метод действия `GetPersonDataJson` в контроллере `People` для подготовки данных, которые мы передаем методу `Json`.

### Листинг 21-20: Подготовка данных для JSON кодировки

```
...
public JsonResult GetPeopleDataJson(string selectedRole = "All") {
    var data = GetData(selectedRole).Select(p => new {
        FirstName = p.FirstName,
        LastName = p.LastName,
        Role = Enum.GetName(typeof(Role), p.Role)
    });
    return Json(data, JsonRequestBehavior.AllowGet);
}
```

Мы использовали LINQ для создания последовательности новых объектов, которые содержат только свойства FirstName и LastName нашего объекта Person и представили значение Role в строковом выражении. В результате этого мы получаем JSON данные, которые содержат только нужные нам свойства, более полезные для нашего jQuery кода:

```
...  
{"FirstName": "Adam", "LastName": "Freeman", "Role": "Admin"}  
...
```

На рисунке 21-9 показаны измененные выходные данные, отображаемые браузером. Естественно, неиспользованные свойства также отправлены, но вы видите, что столбец Role содержит правильные значения.

**Рисунок 21-9:** Результат подготовки данных для JSON кодировки



*Совет*

*Возможно, вам нужно очистить историю в браузере, чтобы увидеть изменения.*

## Обнаружение AJAX запросов в методе действия

Наш контроллер в настоящее время содержит два метода действия, чтобы мы могли поддерживать запросы для HTML и JSON данных. Это, как правило, тот способ, которым мы строим наши контроллеры, потому что нам нравятся много коротких и простых действий, но вы не обязаны работать таким же образом. MVC фреймворк обеспечивает простой способ обнаружения Ajax запросов. Это обозначает, что вы можете создать один метод действия, который управляет разными форматами данных. В листинге 21-21 вы можете увидеть, как мы переделали контроллер Person, чтобы он содержал одно действие, которое обрабатывает как JSON, так и HTML.

**Листинг 21-21:** Создание одного метода действия, который обрабатывает запросы JSON и HTML

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using HelperMethods.Models;
namespace HelperMethods.Controllers
{
    public class PeopleController : Controller
    {
        private Person[] personData = {
            new Person {FirstName = "Adam", LastName = "Freeman", Role = Role.Admin},
            new Person {FirstName = "Steven", LastName = "Sanderson", Role = Role.Admin},
            new Person {FirstName = "Jacqui", LastName = "Griffyth", Role = Role.User},
            new Person {FirstName = "John", LastName = "Smith", Role = Role.User},
            new Person {FirstName = "Anne", LastName = "Jones", Role = Role.Guest}
        };
        public ActionResult Index()
        {
            return View();
        }
        public ActionResult GetPeopleData(string selectedRole = "All")
        {
            IEnumerable<Person> data = personData;
            if (selectedRole != "All")
            {
                Role selected = (Role)Enum.Parse(typeof(Role), selectedRole);
                data = personData.Where(p => p.Role == selected);
            }
            if (Request.IsAjaxRequest())
            {
                var formattedData = data.Select(p => new
                {
                    FirstName = p.FirstName,
                    LastName = p.LastName,
                    Role = Enum.GetName(typeof(Role), p.Role)
                });
                return Json(formattedData, JsonRequestBehavior.AllowGet);
            }
            else
            {
                return PartialView(data);
            }
        }
        public ActionResult GetPeople(string selectedRole = "All")
        {
            return View((object)selectedRole);
        }
    }
}
```

Мы использовали метод Request.IsAjaxRequest для обнаружения Ajax запросов и передачи формата JSON, если результат является true. Есть несколько ограничений, которые вы должны знать прежде, чем будете следовать такому подходу.

Во-первых, метод IsAjaxRequest возвращает true, если браузер включил в свой запрос заголовок X-Requested-With и установил значение на XMLHttpRequest. Это широко используемое соглашение, но оно не является универсальным, и поэтому вы должны рассмотреть тот вариант, будут ли пользователи делать запросы, которые требуют JSON данных, без установки этого заголовка.

Вторым ограничением является то, что предполагается, что все AJAX запросы требуют JSON данных. Возможно, ваше приложение будет лучше поддерживаться, если вы разделите способ, которым был сделан запрос, от формата данных, которые ищет клиент. Мы предпочитаем этот подход, и по этой причине мы склонны определять отдельные методы действий для разных форматов данных.

Нам также нужно внести два изменения в наше представление `GetPerson.cshtml` для поддержки одного метода действия, как показано в листинге 21-22.

**Листинг 21-22:** Изменение представления `GetPerson.cshtml` для поддержки одного метода действия

```
@using HelperMethods.Models
@model string
 @{
    ViewBag.Title = "GetPeople";
    AjaxOptions ajaxOpts = new AjaxOptions {
        Url = Url.Action("GetPeopleData"),
        LoadingElementId = "loading",
        LoadingElementDuration = 1000,
        OnSuccess = "processData"
    };
}
<script type="text/javascript">
    function processData(data) {
        var target = $("#tableBody");
        target.empty();
        for (var i = 0; i < data.length; i++) {
            var person = data[i];
            target.append("<tr><td>" + person.FirstName + "</td><td>" +
                person.LastName + "</td><td>" + person.Role +
                "</td></tr>");
        }
    }
</script>
<h2>Get People</h2>
<div id="loading" class="load" style="display: none">
    <p>Loading Data...</p>
</div>
<table>
    <thead>
        <tr>
            <th>First</th>
            <th>Last</th>
            <th>Role</th>
        </tr>
    </thead>
    <tbody id="tableBody">
        @Html.Action("GetPeopleData", new { selectedRole = Model })
    </tbody>
</table>
@using (Ajax.BeginForm(ajaxOpts)) {
<div>
    @Html.DropDownList("selectedRole", new SelectList(
        new [] {"All"}.Concat(Enum.GetNames(typeof(Role)))))
    <button type="submit">Submit</button>
</div>
}
<div>
    @foreach (string role in Enum.GetNames(typeof(Role))) {
        <div class="ajaxLink">
            @Ajax.ActionLink(role, "GetPeople",
                new { selectedRole = role },
                new { @class = "link" })
        </div>
    }
</div>
```

```

        new AjaxOptions {
            Url = Url.Action("GetPeopleData", new { selectedRole = role}),
            OnSuccess = "processData"
        })
    </div>
}
</div>

```

Первое изменение заключается в объекте `AjaxOptions`, который мы используем для Ajax формы. Поскольку мы уже не можем получить фрагмент HTML через Ajax запрос, мы должны использовать ту же функцию `ProcessData` для обработки JSON данных ответа сервера, которую мы создали для Ajax ссылок. Второе изменение заключается в значении свойства `Url` объектов `AjaxOptions`, которые мы создали для ссылок. Действия `GetPeopleDataJson` больше не существует, вместо этого и мы нацелены на действие `GetPeopleData`.

## Резюме

В этой главе мы рассмотрели ненавязчивый Ajax в MVC, который позволяет нам воспользоваться функциональностью библиотеки JQuery простым и элегантным способом без добавления большого количества кода в представления. Если вы хотите работать с фрагментами HTML, то вам вообще не нужно добавлять какой-либо JavaScript код в представления. Но нам нравится работать с JSON, и это означает, что мы, как правило, работаем с функциями JavaScript, которые используют JQuery для обработки данных и создания нужных HTML элементов. В следующей главе мы рассмотрим один из самых интересных и полезных аспектов MVC фреймворка: модель связывания данных.

# Связывание данных модели

Связывание данных – это процесс создания объектов .NET с помощью данных, отправляемых браузером в запросе HTTP. Мы полагались на связывание данных каждый раз, когда определяли метод действия, который принимает параметр – объекты параметра создаются с помощью связывания данных. В этой главе мы продемонстрируем, как работает система связывания данных, и рассмотрим способы для ее продвинутой настройки.

## Подготовка проекта для примера

Для этой главы мы создали в Visual Studio новый проект MVC под названием `MvcModels` на шаблоне `Basic`. Мы будем использовать тот же класс модели, что и в предыдущих главах, поэтому создайте новый файл под названием `Person.cs` в папке `Models` и приведите его содержимое в соответствие с листингом 22-1.

### Листинг 22-1: Файл /Models/Person.cs

```
using System;

namespace MvcModels.Models
{
    public class Person
    {
        public int PersonId { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime BirthDate { get; set; }
        public Address HomeAddress { get; set; }
        public bool IsApproved { get; set; }
        public Role Role { get; set; }
    }

    public class Address
    {
        public string Line1 { get; set; }
        public string Line2 { get; set; }
        public string City { get; set; }
        public string PostalCode { get; set; }
        public string Country { get; set; }
    }

    public enum Role
    {
        Admin,
        User,
        Guest
    }
}
```

Мы также создали контроллер `Home`, как показано в листинге 22-2. В этом контроллере определено для примера несколько объектов `Person`, а также действие `Index`, которое позволяет выбрать один объект `Person` по значению свойства `PersonId`.

### Листинг 22-2: Контроллер `Home`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
```

```

using System.Web.Mvc;
using MvcModels.Models;

namespace MvcModels.Controllers
{
    public class HomeController : Controller
    {
        private Person[] personData =
        {
            new Person
            {
                PersonId = 1,
                FirstName = "Adam",
                LastName = "Freeman",
                Role = Role.Admin
            },
            new Person
            {
                PersonId = 2,
                FirstName = "Steven",
                LastName = "Sanderson",
                Role = Role.Admin
            },
            new Person
            {
                PersonId = 3,
                FirstName = "Jacqui",
                LastName = "Griffyth",
                Role = Role.User
            },
            new Person
            {
                PersonId = 4,
                FirstName = "John",
                LastName = "Smith",
                Role = Role.User
            },
            new Person
            {
                PersonId = 5,
                FirstName = "Anne",
                LastName = "Jones",
                Role = Role.Guest
            }
        };
        public ActionResult Index(int id)
        {
            Person dataItem = personData.Where(p => p.PersonId == id).First();
            return View(dataItem);
        }
    }
}

```

Для этого метода действия мы создали файл представления под названием `/Views/Home/Index.cshtml`. Содержимое этого файла показано в листинге 22-3. Мы использовали шаблонный вспомогательный метод `DisplayFor`, чтобы отображать значения некоторых свойств модели представления `Person`.

### Листинг 22-3: Файл `/Views/Home/Index.cshtml`

```

@model MvcModels.Models.Person
 @{
    ViewBag.Title = "Index";

```

```

}
<h2>Person</h2>
<div><label>ID:</label>@Html.DisplayFor(m => m.PersonId)</div>
<div><label>First Name:</label>@Html.DisplayFor(m => m.FirstName)</div>
<div><label>Last Name:</label>@Html.DisplayFor(m => m.LastName)</div>
<div><label>Role:</label>@Html.DisplayFor(m => m.Role)</div>

```

Наконец, мы добавили стили CSS в файл /Content/Site.css, как показано в листинге 22-4.

#### Листинг 22-4: Дополнительные стили в файле Site.css

```

label { display: inline-block; width: 100px; font-weight:bold; margin: 5px; }
form label { float: left; }
input.text-box { float: left; margin: 5px; }
button[type=submit] { margin-top: 5px; float: left; clear: left; }
form div {clear: both; }

```

## Понимание модели связывания данных

Связывание данных представляет собой мост между запросом HTTP и методами C#, которые определяют наши методы действий контроллера. Связывание данных используется в той или иной степени в большинстве приложений MVC Framework, в том числе и в том простом примере, который мы создали в предыдущей главе. Чтобы увидеть связывание данных в действии, запустите приложение и перейдите по ссылке /Home/Index/1. Результат показан на рисунке 22-1.

**Рисунок 22-1:** Связывание данных



Наша ссылка содержит значение свойства PersonId объекта Person, который мы хотели отобразить:

/Home/Index/1

Когда MVC Framework вызывает метод Index контроллера Home для обслуживания запроса, он преобразовывает эту часть URL и использует ее в качестве аргумента:

```
public ActionResult Index(int id) {
```

Преобразование сегмента URL в аргумент метода `int` является примером связывания данных. В следующих разделах мы рассмотрим процесс, который был инициирован этим простым примером, а затем изучим более сложные функции связывания данных. Процесс, который запускает связывание данных, начинается после получения запроса и его обработки системой маршрутизации. Мы не изменили для приложения конфигурацию маршрутизации, и для обработки запроса был использован маршрут по умолчанию, который Visual Studio добавила в файл `/App_Start/RouteConfig.cs`. Он показан в листинге 22-5.

**Листинг 22-5:** Маршрут по умолчанию, добавленный в приложение Visual Studio

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace MvcModels
{
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
            routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new
                {
                    controller = "Home",
                    action = "Index",
                    id = UrlParameter.Optional
                }
            );
        }
    }
}
```

Определение и принципы работы маршрутов были описаны в главе 13, и мы не будем повторять эту информацию здесь. Для связывания данных самой важной частью маршрута является дополнительная переменная сегмента `id`. Когда мы переходим по ссылке `/Home/Index/1`, то ее последний сегмент, который указывает на интересующий нас объект `Person`, соотносится с переменной маршрутизации `id`.

Механизм вызова действий, которые мы рассмотрели в главе 15, с помощью информации о маршрутизации определяет, что для обслуживания запроса необходим метод `Index`, но он не может вызвать метод `Index`, если у него нет значений для аргумента метода.

Механизм вызова действий по умолчанию, `ControllerActionInvoker` (см. главу 15), использует механизмы связывания данных (*model binders*) для создания объектов, которые необходимы для вызова действия. Механизмы связывания данных определяются интерфейсом `IModelBinder`, который показан в листинге 22-6. Позже в данной главе мы вернемся к этому интерфейсу, когда будем создавать пользовательский механизм связывания данных.

**Листинг 22-6:** Интерфейс `IModelBinder`

```
namespace System.Web.Mvc
{
    public interface IModelBinder
    {
```

```

    object BindModel(ControllerContext controllerContext,
        ModelBindingContext bindingContext);
    }
}

```

В приложении MVC может быть несколько механизмов связывания данных, и каждый механизм может создавать один или несколько типов моделей. Когда механизму вызова действий нужно вызвать какой-либо метод действия, он смотрит на параметры, которые определяют данный метод, и находит механизм связывания данных для типа каждого параметра. В нашем примере механизм вызова действий обнаружит, что метод `Index` имеет только один параметр `int`. Затем он найдет механизм связывания, работающий со значениями `int`, и вызовет его метод `BindModel`.

Механизм связывания должен предоставить значение `int`, так чтобы его можно было использовать для вызова метода `Index`. Обычно для этого он трансформирует какой-либо элемент данных запроса (например, значения формы или строковые значения запроса), но в MVC Framework нет никаких ограничений на то, как будут получены данные.

Далее в этой главе мы покажем вам несколько примеров пользовательских механизмов связывания. Мы также продемонстрируем вам некоторые функции класса `ModelBindingContext`, который передается в метод `IModelBinder.BindModel`.

## Использование модели связывания данных по умолчанию

Пользовательские механизмы связывания в приложении определяются редко, чаще всего используется встроенный класс механизма связывания, `DefaultModelBinder`. Он используется механизмом вызова действий, когда отсутствует пользовательский механизм связывания для данного типа. По умолчанию этот механизм связывания проверяет четыре места, которые показаны в таблице 22-1, чтобы найти данные, соответствующие имени параметра, для которого выполняется связывание.

**Таблица 22-1:** Порядок поиска данных для параметра классом `DefaultModelBinder`

Место	Описание
<code>Request.Form</code>	Значения, предоставленные пользователем в элементе HTML <code>form</code> .
<code>RouteData.Values</code>	Значения, полученные через маршруты приложения.
<code>Request.QueryString</code>	Данные строки запроса из URL.
<code>Request.Files</code>	Файлы, загруженные как часть запроса (пример загрузки файлов показан в главе 11).

Места проверяются в указанном порядке. Так, в нашем примере `DefaultModelBinder` будет искать значение параметра `id` следующим образом:

1. `Request.Form["id"]`
2. `RouteData.Values["id"]`
3. `Request.QueryString["id"]`
4. `Request.Files["id"]`

Как только значение найдено, поиск будет остановлен. В нашем примере поиск по данным формы не дал результата, но требуемое имя было найдено в переменной маршрутизации. Это означает, что поиск в строке запроса и загруженных файлах не будет проводиться вообще.

## Подсказка

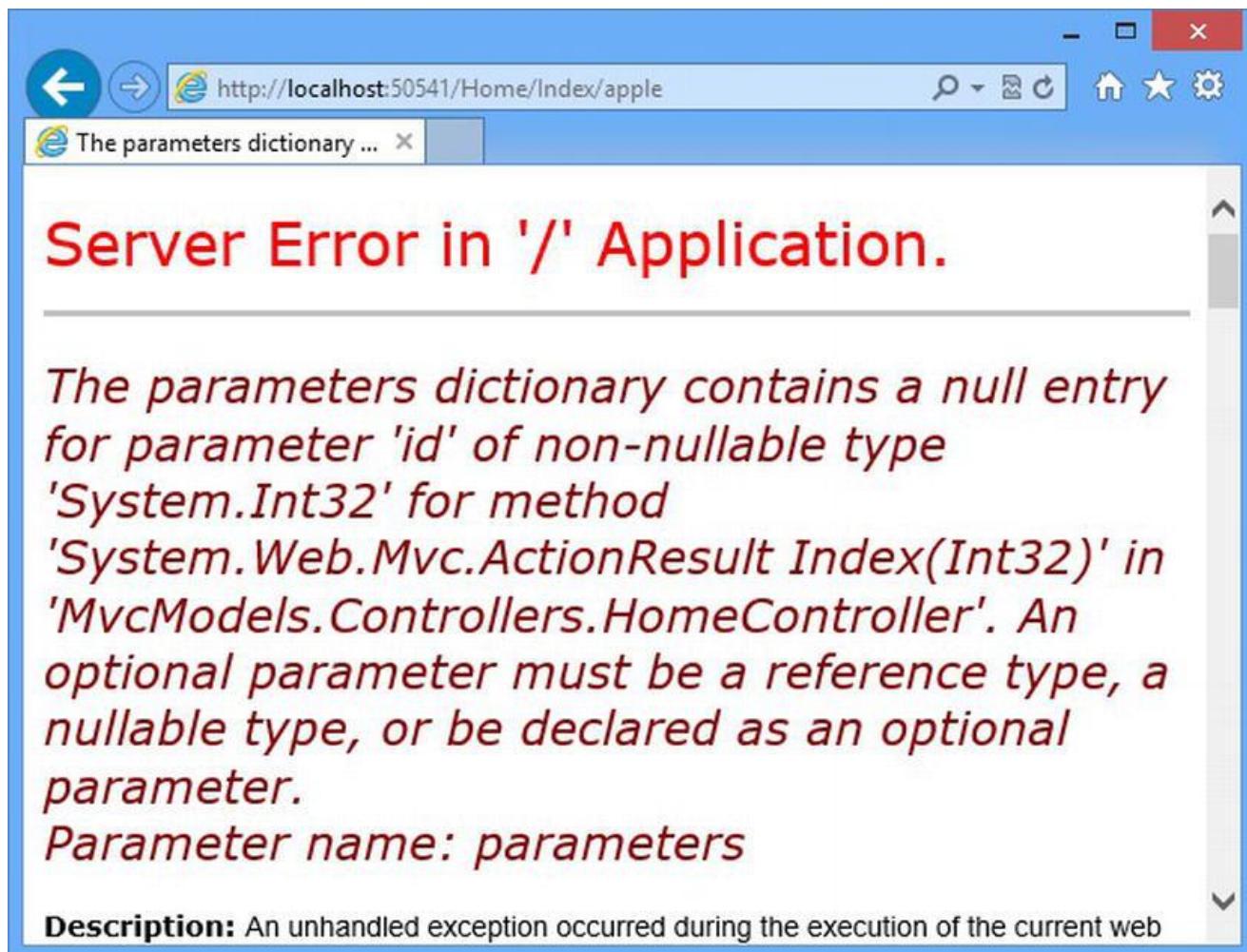
Если вы используете механизм связывания по умолчанию, важно, чтобы параметры вашего метода действия соответствовали названиям свойств. Наш пример приложения работает, потому что имя параметра метода действия соответствует имени переменной маршрутизации. Если параметр будет называться, к примеру, `personId`, механизм связывания по умолчанию не сможет найти соответствующее значение данных, и запрос не будет обработан.

## Связывание простых типов

Для простых типов параметров DefaultModelBinder пытается преобразовать полученное из данных запроса строковое значение в тип параметра с помощью класса `System.ComponentModel.TypeDescriptor`. Если значение не может быть преобразовано, например, если мы предоставили значение `apple` для параметра, который требует значение `int`, то DefaultModelBinder не сможет связать данные.

Чтобы увидеть возникающую из-за этого проблему, запустите приложение и перейдите по ссылке `/Home/Index/apple`. На рисунке 22-2 показан ответ от сервера.

Рисунок 22-2: Ошибка при обработке свойства модели



Механизм связывания по умолчанию упорно пытается привести предоставленное нами значение `apple` к требуемому типу `int`, что вызывает ошибку, показанную на рисунке.

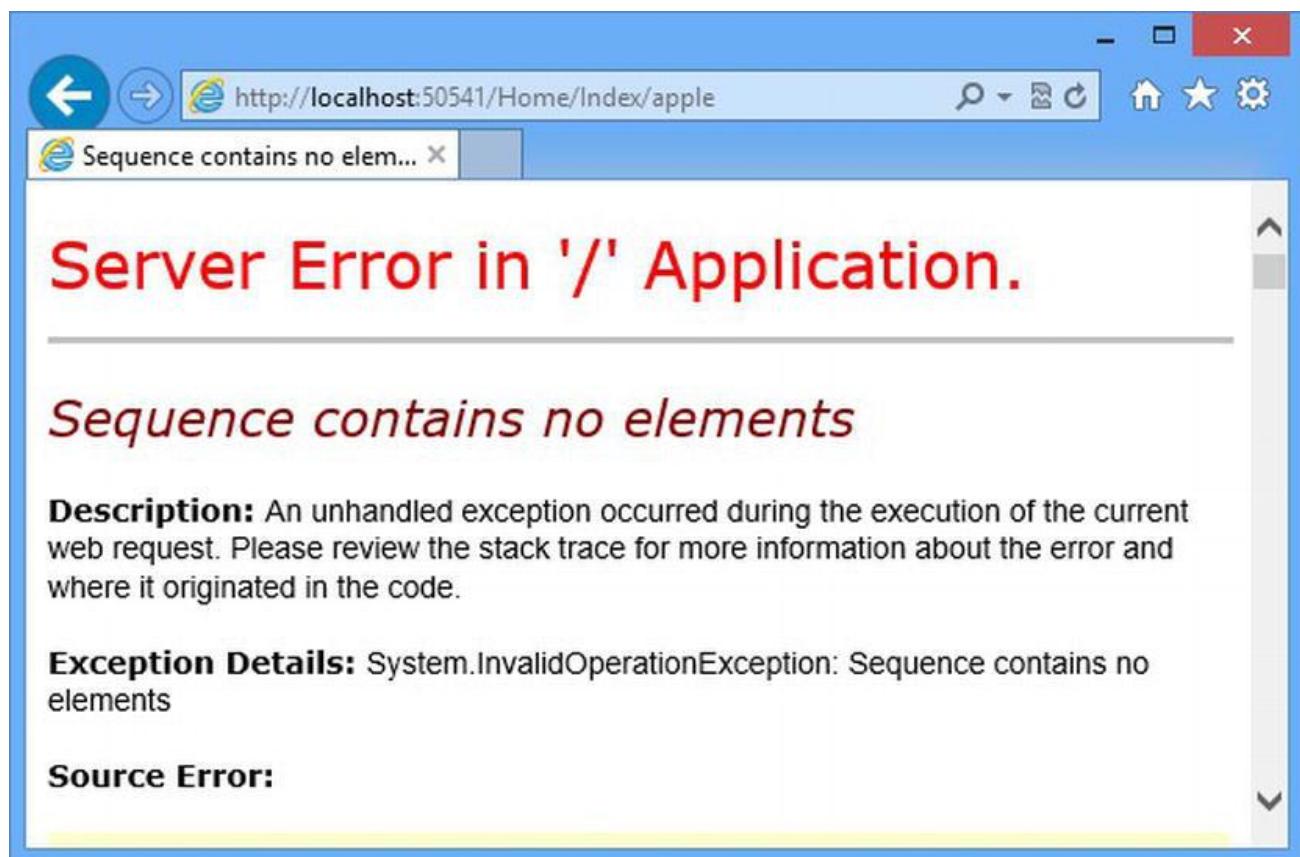
Упростить работу механизма связывания можно с помощью типов, допускающих значение `null`, которые обеспечивают запасную позицию. Вместо того чтобы требовать числовое значение, параметр `int?` позволяет механизму связывания установить аргументу метода действия значение `null`. В листинге 22-7 показано, как мы применили тип, допускающий значение `null`, к действию `Index`.

**Листинг 22-7:** Используем тип, допускающий значение `null`, для параметра метода действия

```
public ActionResult Index(int? id)
{
    Person dataItem = personData.Where(p => p.PersonId == id).First();
    return View(dataItem);
}
```

Если вы запустите приложение и перейдите по ссылке `/Home/Index/apple`, то увидите, что мы решили только часть проблемы, как показано на рисунке 22-3.

**Рисунок 22-3:** Запрос для значения `null`



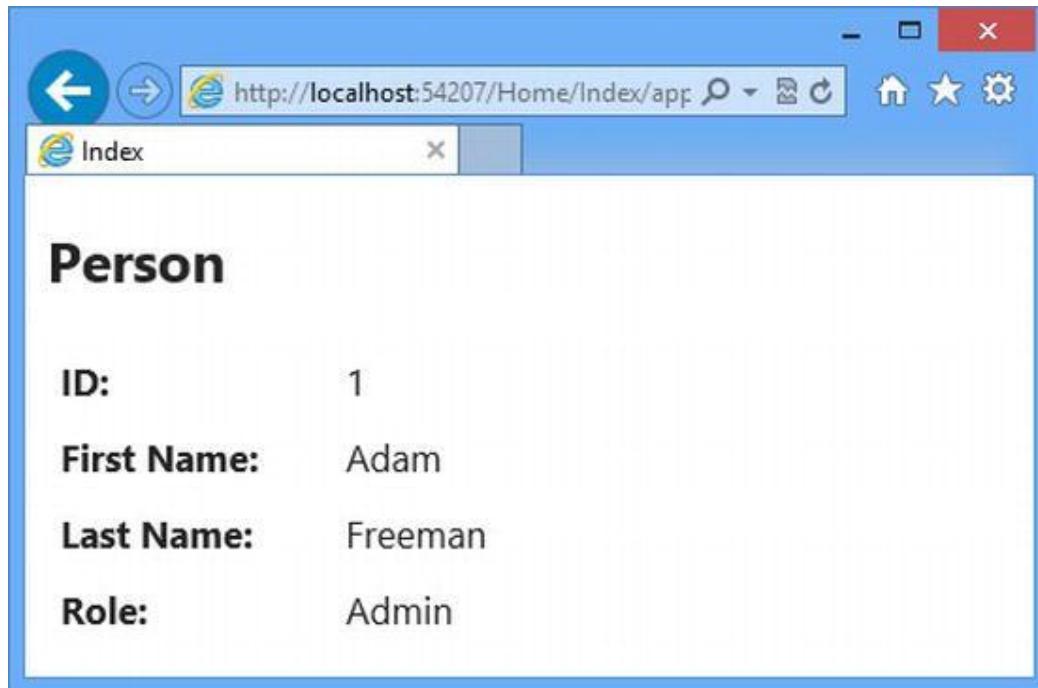
Мы изменили характер проблемы – механизм связывания может использовать `null` в качестве значения аргумента `id` метода `Index`, но код метода действия не обрабатывает значение `null`. Это можно было бы исправить, явно включив в метод код для обработки значений `null`, но мы установили для параметра значение по умолчанию, которое будет использоваться вместо `null`. В листинге 22-8 показано, как мы применили значение параметра по умолчанию в методе действия `Index`.

### Листинг 22-8: Применяем значение параметра по умолчанию в методе действия Index

```
public ActionResult Index(int id = 1)
{
    Person dataItem = personData.Where(p => p.PersonId == id).First();
    return View(dataItem);
}
```

Всякий раз, когда механизм связывания не сможет найти значение для параметра `id`, будет использоваться значение по умолчанию 1, что равнозначно выбору объекта `Person` со значением свойства `PersonId` 1, как показано на рисунке 22-4.

Рисунок 22-4: Результат использования значения параметра по умолчанию в методе действий



#### Подсказка

Имейте в виду, что мы решили проблему нечисловых значений для механизма связывания, но мы все еще можем получить значения `int`, для которых в контроллере `Home` нет действительных объектов `Person`. Например, механизм связывания может запросить преобразование последний сегмент URL-адресов `/Home/Index/-1` и `/Home/Index/500` в значения `int`. Это предоставит методу действия действительное значение для вызова метода `Index`, но приведет к ошибке, потому что наш код не выполняет никаких дополнительных проверок. Рекомендуем вам обратить внимание на диапазон значений параметров, с которым может работать ваш метод действия, и создать соответствующие тесты.

### Анализ с учетом настроек культуры

Класс `DefaultModelBinder` использует различные настройки культуры для преобразования типов из разных областей данных запроса. Значения, получаемые из URL (данные маршрутизации и строки запроса), преобразуются без учета настроек культуры (*culture-insensitive parsing*), но эти настройки принимаются во внимание для значений из данных форм.

Чаще всего это вызывает проблемы со значениями `DateTime`. Даты, которые обрабатываются без учета настроек культуры, должны быть в универсальном формате `yyyy-mm-dd`. Значения дат из форм должны быть в формате, указанном сервером. Это означает, что на сервере с настройками культуры `uk` формат даты будет `dd-mm-yyyy`, в то время как на сервере с настройками культуры `us` формат будет `mm-dd-yyyy`, хотя `yyyy-mm-dd` будет приемлемым в обоих случаях.

Если значение даты не соответствует формату, оно не будет преобразовано. Следовательно, мы должны убедиться, что все даты в URL соответствуют универсальному формату записи. Мы также должны быть осторожны при обработке значений дат, предоставленных пользователями. Механизм связывания по умолчанию предполагает, что пользователи будут записывать даты в соответствии с форматом, указанном в настройках культуры сервера, что вряд ли будет соответствовать действительности, если приложение будет использоваться клиентами из разных стран.

## Связывание сложных типов

Когда параметр метода действия является сложным типом (т.е. типом, который не может быть преобразован с помощью класса `TypeConverter`), то `DefaultModelBinder` класс использует *рефлексию (reflection)*, чтобы получить список общих свойств, а затем связывает их по очереди. Для демонстрации, как это работает, мы добавили два новых метода действий в контроллер `Home`, как показано в листинге 22-9.

**Листинг 22-9:** Добавляем новые методы действий в контроллер `Home`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using MvcModels.Models;

namespace MvcModels.Controllers
{
    public class HomeController : Controller
    {
        private Person[] personData =
        {
            new Person
            {
                PersonId = 1,
                FirstName = "Adam",
                LastName = "Freeman",
                Role = Role.Admin
            },
            new Person
            {
                PersonId = 2,
                FirstName = "Steven",
                LastName = "Sanderson",
                Role = Role.Admin
            },
            new Person
            {
                PersonId = 3,
                FirstName = "Jacqui",
                LastName = "Griffyth",
                Role = Role.User
            },
            new Person
            {
                PersonId = 4,
```

```

        FirstName = "John",
        LastName = "Smith",
        Role = Role.User
    },
    new Person
    {
        PersonId = 5,
        FirstName = "Anne",
        LastName = "Jones",
        Role = Role.Guest
    }
};

public ActionResult Index(int id = 1)
{
    Person dataItem = personData.Where(p => p.PersonId == id).First();
    return View(dataItem);
}

public ActionResult CreatePerson()
{
    return View(new Person());
}

[HttpPost]
public ActionResult CreatePerson(Person model)
{
    return View("Index", model);
}
}
}
}

```

Перегруженная версия метода `CreatePerson` без параметров создает новый объект `Person` и передает его в метод представления, в результате чего визуализируется представление `Views/Home/CreatePerson.cshtml`, которое показано в листинге 22-10.

#### Листинг 22-10: Представление `CreatePerson.cshtml`

```

@model MvcModels.Models.Person
 @{
    ViewBag.Title = "CreatePerson";
}
<h2>Create Person</h2>
@using (Html.BeginForm())
{
    <div>@Html.LabelFor(m => m.PersonId)@Html.EditorFor(m => m.PersonId)</div>
    <div>@Html.LabelFor(m => m.FirstName)@Html.EditorFor(m => m.FirstName)</div>
    <div>@Html.LabelFor(m => m.LastName)@Html.EditorFor(m => m.LastName)</div>
    <div>@Html.LabelFor(m => m.Role)@Html.EditorFor(m => m.Role)</div>
    <button type="submit">Submit</button>
}

```

Это представление визуализирует простой набор элементов `label` и `editor` для свойств объекта `Person`, с которыми мы работаем, а также элемент `form`, который отправляет данные элементов `editor` к методу действия `CreatePerson` с атрибутом `HttpPost`. Он использует представление `Views/Home/Index.cshtml` для отображения данных, переданных в форме. Чтобы увидеть, как работают новые методы действий, запустите приложение и перейдите по ссылке `/Home/CreatePerson`, как показано на рисунке 22-5.

**Рисунок 22-5:** Используем методы действий CreatePerson



Отправляя форму к методу `CreatePerson`, мы создаем различные ситуации для связывания данных. Механизм связывания по умолчанию обнаруживает, что наш метод действия требует объект `Person` и обрабатывает все его свойства по очереди. Для каждого свойства простого типа механизм связывания пытается найти значение запроса, как и в предыдущем примере. Так, например, обнаружив свойство `PersonID`, он будет искать значение данных `PersonID`, которые найдет в данных формы в запросе.

Если для свойства требуется сложный тип, то для нового типа процесс повторяется - будут получены общедоступные свойства и механизм связывания попытается найти значения для каждого из них. Разница в том, что имена свойств являются вложенными. Например, свойство `HomeAddress` класса `Person` принадлежит типу `Address`, который показан в листинге 22-11.

#### Листинг 22-11: Вложенный класс модели

```
public class Address
{
    public string Line1 { get; set; }
    public string Line2 { get; set; }
    public string City { get; set; }
    public string PostalCode { get; set; }
    public string Country { get; set; }
}
```

Чтобы найти значение для свойства `Line1`, механизм связывания будет искать значение для `HomeAddress.Line1`, то есть, он объединит имя свойства, указанное в объекте модели, с именем свойства, указанным в типе свойства.

#### Создаем легко связываемый HTML

Использование строк с префиксами означает, что мы должны создавать представления, которые будут их визуализировать - хотя вспомогательные методы значительно упрощают эту задачу. В листинге 22-12 показано, как мы обновили файл представления `CreatePerson.cshtml`, чтобы использовать несколько свойств из типа `Address`.

#### Листинг 22-12: Обновляем представление `CreatePerson.cshtml`, чтобы использовать свойства из типа `Address`

```
@model MvcModels.Models.Person
{@
```

```

ViewBag.Title = "CreatePerson";
}
<h2>Create Person</h2>
@using (Html.BeginForm())
{
    <div>@Html.LabelFor(m => m.PersonId)@Html.EditorFor(m => m.PersonId)</div>
    <div>@Html.LabelFor(m => m.FirstName)@Html.EditorFor(m => m.FirstName)</div>
    <div>@Html.LabelFor(m => m.LastName)@Html.EditorFor(m => m.LastName)</div>
    <div>@Html.LabelFor(m => m.Role)@Html.EditorFor(m => m.Role)</div>
    <div>
        @Html.LabelFor(m => m.HomeAddress.City)
        @Html.EditorFor(m => m.HomeAddress.City)
    </div>
    <div>
        @Html.LabelFor(m => m.HomeAddress.Country)
        @Html.EditorFor(m => m.HomeAddress.Country)
    </div>
    <button type="submit">Submit</button>
}

```

Мы использовали строго типизированный вспомогательный метод `EditorFor` и указали свойства, которые мы хотим отредактировать, из свойства `HomeAddress`. Вспомогательный метод автоматически задает атрибуты `name` элементов `input` в соответствии с форматом, который используется механизмом связывания по умолчанию, например:

```
<input class="text-box single-line" id="HomeAddress_Country" name="HomeAddress.Country"
      type="text" value="" />
```

Благодаря этой функции нам не нужно самим проверять, сможет ли механизм связывания создать объект `Address` для свойства `HomeAddress`. Чтобы увидеть, как это работает, мы отредактируем представление `/Views/Home/Index.cshtml`, в котором будем отображать свойства `HomeAddress`, полученные из формы, как показано в листинге 22-13.

### Листинг 22-13: Отображаем свойства `HomeAddress.City` и `HomeAddress.Country`

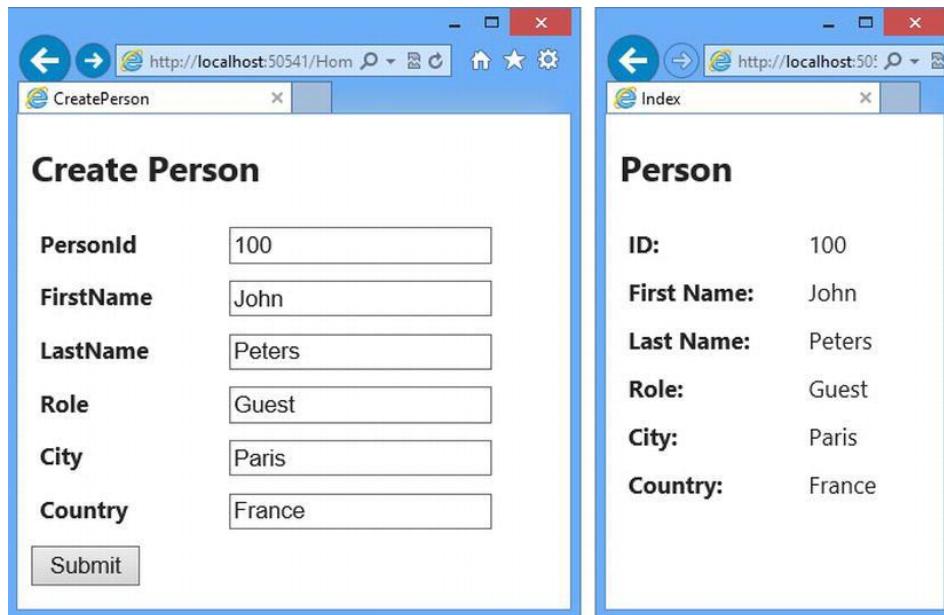
```

@model MvcModels.Models.Person
 @{
    ViewBag.Title = "Index";
}
<h2>Person</h2>
<div><label>ID:</label>@Html.DisplayFor(m => m.PersonId)</div>
<div><label>First Name:</label>@Html.DisplayFor(m => m.FirstName)</div>
<div><label>Last Name:</label>@Html.DisplayFor(m => m.LastName)</div>
<div><label>Role:</label>@Html.DisplayFor(m => m.Role)</div>
<div><label>City:</label>@Html.DisplayFor(m => m.HomeAddress.City)</div>
<div><label>Country:</label>@Html.DisplayFor(m => m.HomeAddress.Country)</div>

```

Запустите приложение и перейдите по ссылке `/Home/CreatePerson`, введите значения для свойств `City` и `Country` и убедитесь, что при отправке формы они связываются с объектом модели, как показано на рисунке 22-6.

**Рисунок 22-6:** Связывание свойств в сложных объектах



### Определяем пользовательские префиксы

Бывают ситуации, когда HTML, который вы генерируете, относится к одному типу объекта, но вы хотите связать его с другим. Это означает, что префиксы, содержащие представление, не будут соответствовать структуре, которую ожидает механизм связывания, и ваши данные не будут обработаны должным образом.

Чтобы продемонстрировать эту ситуацию, мы создали новый класс под названием `AddressSummary.cs` в папке `Models`. Содержимое этого файла показано в листинге 22-14.

#### Листинг 22-14: Класс AddressSummary.cs

```
namespace MvcModels.Models
{
    public class AddressSummary
    {
        public string City { get; set; }
        public string Country { get; set; }
    }
}
```

Мы добавили новый метод действия в контроллер `Home`, который использует класс `AddressSummary`; он показан в листинге 22-15.

#### Листинг 22-15: Добавляем новый метод действия в контроллер `Home`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using MvcModels.Models;

namespace MvcModels.Controllers
{
    public class HomeController : Controller
    {
        // ...other statements omitted from listing for brevity...
    }
}
```

```

public ActionResult DisplaySummary(AddressSummary summary)
{
    return View(summary);
}
}

```

Наш новый метод называется `DisplaySummary`. Он принимает параметр `AddressSummary`, который передает в метод `View`. Представление для этого метода называется `DisplaySummary.cshtml`. Мы создали его в папке `/views/Home`. Содержимое этого представления показано в листинге 22-16.

### Листинг 22-16: Представление `DisplaySummary.cshtml`

```

@model MvcModels.Models.AddressSummary
 @{
    ViewBag.Title = "DisplaySummary";
}
<h2>Address Summary</h2>
<div><label>City:</label>@Html.DisplayFor(m => m.City)</div>
<div><label>Country:</label>@Html.DisplayFor(m => m.Country)</div>

```

Это очень простое представление, которое просто отображает значения двух свойств, определенных в классе `AddressSummary`. Чтобы продемонстрировать проблему с префиксами при связывании с другими типами моделей, мы изменим вызов к вспомогательному методу файла `/Views/Home/CreatePerson.cshtml`, чтобы форма была отправлена к новому методу действия `DisplaySummary`, как показано в листинге 22-17.

### Листинг 22-17: Изменяем метод для отправки формы в представлении `CreatePerson.cshtml`

```

@model MvcModels.Models.Person
 @{
    ViewBag.Title = "CreatePerson";
}
<h2>Create Person</h2>
@using (Html.BeginForm("DisplaySummary", "Home"))
{
    <div>@Html.LabelFor(m => m.PersonId)@Html.EditorFor(m => m.PersonId)</div>
    <div>@Html.LabelFor(m => m.FirstName)@Html.EditorFor(m => m.FirstName)</div>
    <div>@Html.LabelFor(m => m.LastName)@Html.EditorFor(m => m.LastName)</div>
    <div>@Html.LabelFor(m => m.Role)@Html.EditorFor(m => m.Role)</div>
    <div>
        @Html.LabelFor(m => m.HomeAddress.City)
        @Html.EditorFor(m => m.HomeAddress.City)
    </div>
    <div>
        @Html.LabelFor(m => m.HomeAddress.Country)
        @Html.EditorFor(m => m.HomeAddress.Country)
    </div>
    <button type="submit">Submit</button>
}

```

Чтобы увидеть проблему, запустите приложение и перейдите по ссылке `/Home/CreatePerson`. При отправке формы значения, введенные для свойств `City` и `Country`, не отображаются в HTML представления `DisplaySummary`.

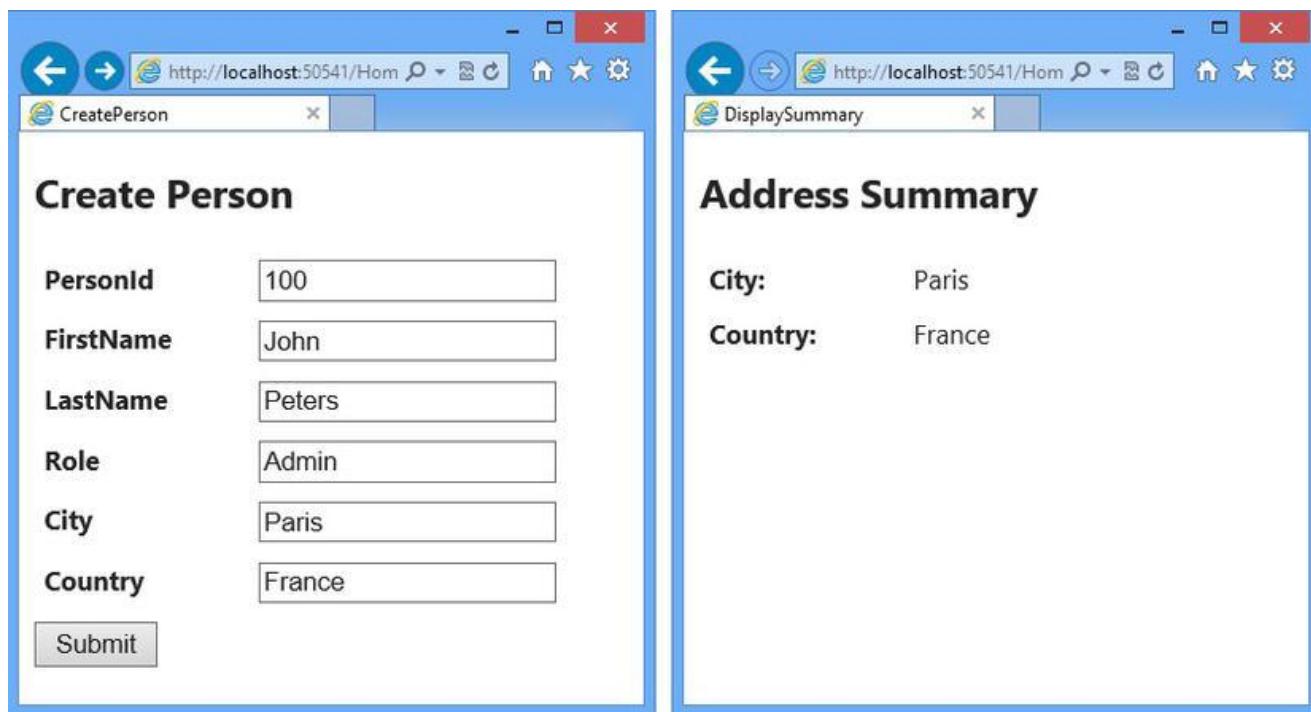
Проблема в том, что атрибуты `name` в форме имеют префикс `HomeAddress`, а это не то, что ищет механизм связывания, когда пытается связать тип `AddressSummary`. Чтобы исправить это, можно применить к параметру метода действия атрибут `Bind`, который сообщает механизму связывания, какой префикс он должен искать, как показано в листинге 22-18.

### Листинг 22-18: Применяем атрибут Bind к параметру метода действия

```
public ActionResult DisplaySummary([Bind(Prefix = "HomeAddress")] AddressSummary summary)
{
    return View(summary);
}
```

Синтаксис запутанный, но результат его оправдывает. Заполняя свойства объекта AddressSummary, механизм связывания будет искать в запросе данные значений HomeAddress.City и HomeAddress.Country. В этом примере мы отображаем элементы editor для свойств объекта Person, но после отправки формы создали экземпляр класса AddressSummary с помощью механизма связывания, как показано на рисунке 22-7. Этот способ может показаться слишком объемным для решения такой простой проблемы, но связывание с другим объектом используется очень часто, и, скорее всего, вы будете использовать эту технику в реальных проектах.

Рисунок 22-7: Связывание свойств с другим типом объекта



### Выборочное связывание свойств

Представьте, что в свойство Country класса AddressSummary является особенно чувствительным, и мы не хотим, чтобы пользователи могли указывать для него значения. Чтобы не отображать данное свойство для пользователей или даже запретить включать его в HTML для отправки в браузер, можно использовать атрибуты, что мы рассмотрели в главе 20, или просто не добавлять для этого свойства элементы editor в представлении.

Тем не менее, злоумышленник может просто отредактировать данные формы, которые отправляются на сервер при отправке формы, и выбрать значение для свойства Country, которое ему подходит. На самом деле, мы хотим сообщить механизму связывания, чтобы он не связывал значение для свойства Country из запроса, для чего нужно применить к параметру метода действия атрибут Bind. В листинге 22-19 показано, как с помощью атрибута мы запретили пользователю предоставлять значение для свойства Country в методе действия DisplaySummary контроллера Home.

### Листинг 22-19: Запрещаем связывание данных для свойства

```
public ActionResult DisplaySummary(
    [Bind(Prefix = "HomeAddress", Exclude = "Country")] AddressSummary summary)
{
    return View(summary);
}
```

Свойство `Exclude` атрибута `Bind` позволяет исключить свойства из процесса связывания данных. Чтобы увидеть эффект, перейдите по ссылке `/Home/CreatePerson`, введите данные и отправьте форму. Вы увидите, что для свойства `Country` ничего не отображается. (В качестве альтернативы, вы можете использовать свойство `Include`, чтобы указать только те свойства, которые должны быть связаны для этой модели; все остальные свойства будут игнорироваться).

Когда к параметру метода действия применяется атрибут `Bind`, он влияет только на те экземпляры класса, которые связываются для данного метода действия; все остальные методы действий будут пытаться выполнить связывание для всех свойств, определенных типом параметра. Если вы хотите получить более широкий результат, примените атрибут `Bind` к самому классу модели, как показано в листинге 22-20. Здесь мы применили метод `Bind` к классу `AddressSummary`, чтобы включить в процесс связывания только свойство `City`.

### Листинг 22-20: Применяем атрибут `Bind` к классу модели

```
using System.Web.Mvc;

namespace MvcModels.Models
{
    [Bind(Include = "City")]
    public class AddressSummary
    {
        public string City { get; set; }
        public string Country { get; set; }
    }
}
```

Когда атрибут `Bind` применен к классу модели *и* параметру метода действия, свойство будет включено в процесс связывания только в том случае, если ни один атрибут его не исключает. Это означает, что если мы исключаем свойство в классе модели, а в параметре метода действия уже используем менее ограничительные правила, это свойство все равно будет исключено из процесса связывания данных.

## Связывание с массивами и коллекциями

Механизм связывания по умолчанию включает поддержку связывания данных запроса с массивами и коллекциями. Мы продемонстрируем эти функции в следующих разделах, после чего покажем вам, как настроить процесс связывания данных.

### Связывание с массивами

В механизме связывания по умолчанию очень хорошо реализована поддержка параметров методов действий, которые представляют собой массивы. Чтобы ее продемонстрировать, мы добавили в контроллер `Home` новый метод под названием `Names`, который показан в листинге 22-21.

### Листинг 22-21: Добавляем метод `Names` в контроллер `Home`

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```

using System.Web;
using System.Web.Mvc;
using MvcModels.Models;

namespace MvcModels.Controllers
{
    public class HomeController : Controller
    {
        // ...other statements omitted from listing for brevity
        public ActionResult Names(string[] names)
        {
            names = names ?? new string[0];
            return View(names);
        }
    }
}

```

Метод действия `Names` принимает в качестве параметра массив строк под названием `names`. Механизм связывания будет искать любые данные с именем `names` и создаст массив, который будет содержать эти значения.

#### *Подсказка*

*Обратите внимание, что в этом примере мы проверяем, не содержит ли параметр `null` в данном методе действия. В качестве значение по умолчанию для параметров можно использовать только константы или литералы.*

---

В листинге 22-22 показано представление `/Views/Home/Names.cshtml`, с помощью которого продемонстрируем связывание массивов.

**Листинг 22-22:** Содержание файла `Names.cshtml`

```

@model string[]
@{
    ViewBag.Title = "Names";
}
<h2>Names</h2>
@if (Model.Length == 0)
{
    using (Html.BeginForm())
    {
        for (int i = 0; i < 3; i++)
        {
            <div><label>@(i + 1):</label>@Html.TextBox("names")</div>
        }
        <button type="submit">Submit</button>
    }
}
else
{
    foreach (string str in Model)
    {
        <p>@str</p>
    }
    @Html.ActionLink("Back", "Names");
}

```

Это представление отображает различный контент в зависимости от количества элементов в модели представления. Если элементы отсутствуют, то мы выводим форму с тремя одинаковыми элементами ввода:

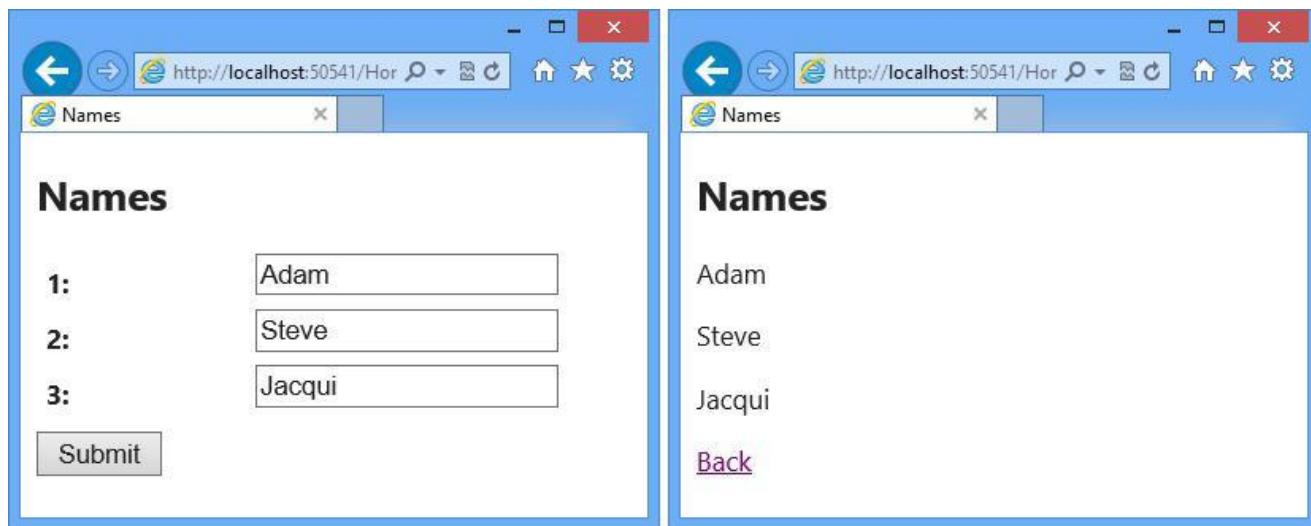
```

<form action="/Home/Names" method="post">
    <div><label>1:</label><input id="names" name="names" type="text" value="" /></div>
    <div><label>2:</label><input id="names" name="names" type="text" value="" /></div>
    <div><label>3:</label><input id="names" name="names" type="text" value="" /></div>
    <button type="submit">Submit</button>
</form>

```

Когда мы отправим форму, механизм связывания по умолчанию увидит, что для нашего метода действия требуется массив строк, и будет искать данные с тем же именем, что и параметр. В нашем примере это означает, что содержание всех наших элементов `input` будет собрано вместе. На рисунке 22-8 показано, как работают метод действия и представление.

**Рисунок 22-8:** Связывание данных для массивов



### Связывание с коллекциями

Мы можем выполнять связывание не только с массивами, но также и с классами коллекций .NET. В листинге 22-23 показано, как мы изменили тип параметра метода действия `Names` на строго типизированный список.

**Листинг 22-23:** Используем строго типизированную коллекцию в качестве параметра метода действия

```

public ActionResult Names(IList<string> names)
{
    names = names ?? new List<string>();
    return View(names);
}

```

Обратите внимание, что мы использовали интерфейс `IList`. Нам не нужно даже указывать конкретную реализацию класса (хотя при желании это можно сделать). В листинге 22-24 показано, как мы изменили файл `Names.cshtml`, чтобы использовать новый тип модели.

**Листинг 22-24:** Используем коллекцию в качестве типа модели в представлении `Names.cshtml`

```

@model IList<string>
 @{
    ViewBag.Title = "Names";
}
<h2>Names</h2>
@if (Model.Count == 0)
{

```

```

using (Html.BeginForm())
{
    for (int i = 0; i < 3; i++)
    {
        <div><label>@(i + 1):</label>@Html.TextBox("names")</div>
    }
    <button type="submit">Submit</button>
}
}
else
{
    foreach (string str in Model)
    {
        <p>@str</p>
    }
    @Html.ActionLink("Back", "Names");
}

```

Функциональность действия Names не изменилась, но теперь мы можем работать с классом коллекции, а не массивом.

## Связывание с коллекциями пользовательских типов моделей

Мы также можем связать отдельные свойства данных с массивом пользовательских типов, таким как наш класс модели AddressSummary. В листинге 22-25 показано, что мы добавили новый метод действия под названием Address, который принимает в качестве параметров строго типизированную коллекцию, которая полагается на пользовательский класс модели.

### Листинг 22-25: Определяем метод действия с коллекцией пользовательских типов модели

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using MvcModels.Models;

namespace MvcModels.Controllers
{
    public class HomeController : Controller
    {
        // ...other statements omitted from listing for brevity...
        public ActionResult Address(IList<AddressSummary> addresses)
        {
            addresses = addresses ?? new List<AddressSummary>();
            return View(addresses);
        }
    }
}

```

Для этого метода действия мы создали представление /Views/Home/Address.cshtml, которое показано в листинге 22-26.

### Листинг 22-26: Содержание файла Address.cshtml

```

@using MvcModels.Models
@model IList<AddressSummary>
 @{
    ViewBag.Title = "Address";
}
<h2>Addresses</h2>
@if (Model.Count() == 0)

```

```

{
    using (Html.BeginForm())
    {
        for (int i = 0; i < 3; i++)
        {
            <fieldset>
                <legend>Address @(i + 1)</legend>
                <div><label>City:</label>@Html.Editor("[ " + i + " ].City")</div>
                <div><label>Country:</label>@Html.Editor("[ " + i + " ].Country")</div>
            </fieldset>
        }
        <button type="submit">Submit</button>
    }
}
else
{
    foreach (AddressSummary str in Model)
    {
        <p>@str.City, @str.Country</p>
    }
    @Html.ActionLink("Back", "Address");
}

```

Если в коллекции модели нет элементов, это представление визуализирует элемент `form`. Форма состоит из пар элементов `input`, атрибуты `name` которых начинаются с индекса массива:

```

<fieldset>
    <legend>Address 1</legend>
    <div>
        <label>City:</label>
        <input class="text-box single-line" name="[0].City" type="text" value="" />
    </div>
    <div>
        <label>Country:</label>
        <input class="text-box single-line" name="[0].Country" type="text" value="" />
    </div>
</fieldset>
<fieldset>
    <legend>Address 2</legend>
    <div>
        <label>City:</label>
        <input class="text-box single-line" name="[1].City" type="text" value="" />
    </div>
    <div>
        <label>Country:</label>
        <input class="text-box single-line" name="[1].Country" type="text" value="" />
    </div>
</fieldset>

```

После того, как форма отправлена, механизм связывания по умолчанию обнаружит, что ему нужно создать коллекцию объектов `AddressSummary`, и будет использовать префиксы индексов массива из атрибутов `name`, чтобы получить значения для свойств объекта. Свойства с префиксом `[0]` используются для первого объекта `AddressSummary`, с префиксом `[1]` - для второго объекта, и так далее.

Наше представление `Address.cshtml` определяет элементы `input` для трех таких объектов с индексами и отображает их, когда в коллекции модели есть элементы. Прежде чем мы сможем это продемонстрировать, мы должны удалить атрибут `Bind` из класса модели `AddressSummary`, как показано в листинге 22-27, в противном случае механизм связывания проигнорирует свойство `Country`.

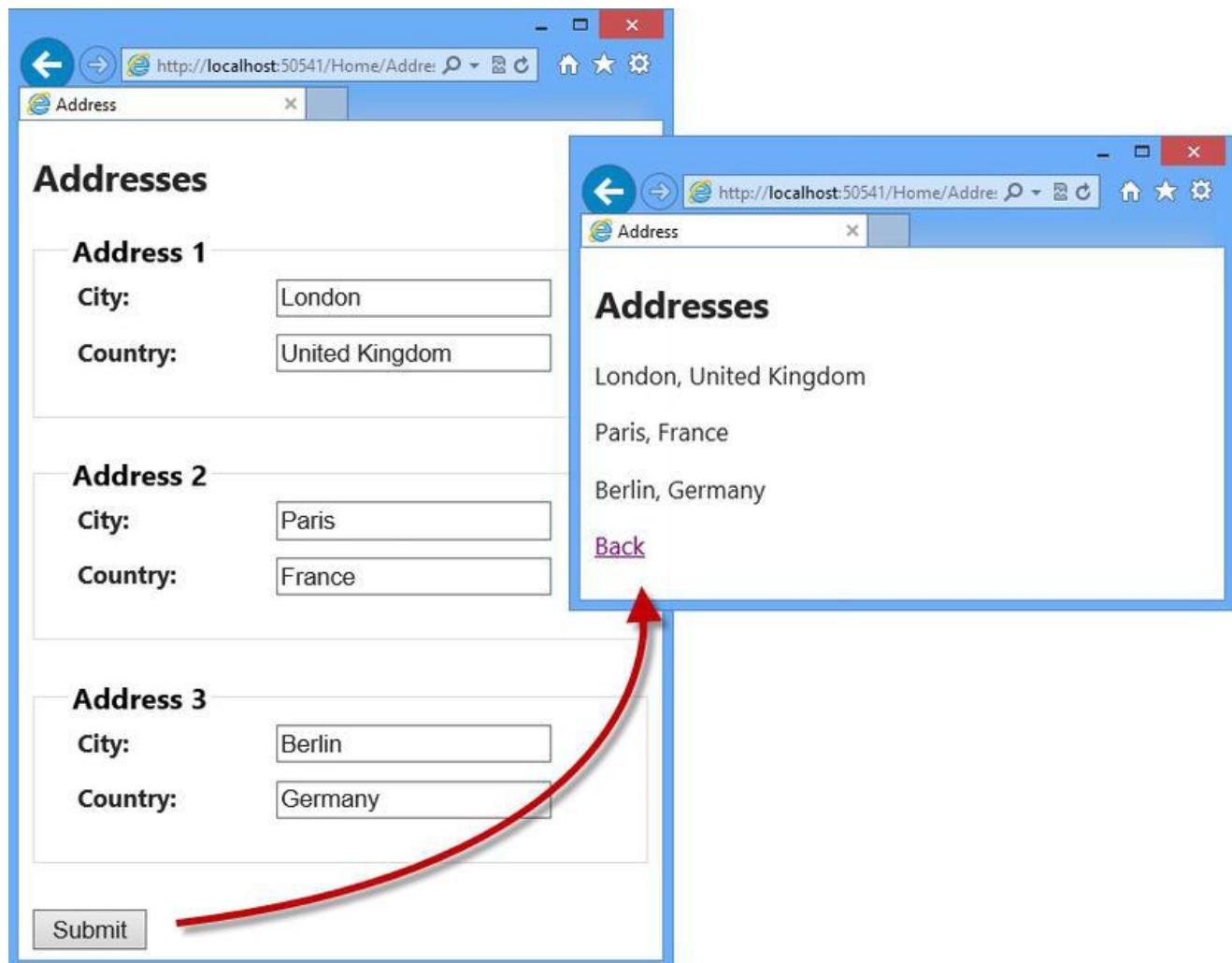
**Листинг 22-27:** Удаляем атрибут Bind из класса модели AddressSummary

```
using System.Web.Mvc;

namespace MvcModels.Models
{
    // This attribute has been commented out
    // [Bind(Include="City")]
    public class AddressSummary
    {
        public string City { get; set; }
        public string Country { get; set; }
    }
}
```

Чтобы увидеть, как процесс связывания данных работает для пользовательских коллекций объектов, запустите приложение и перейдите по ссылке /Home/Address. Введите несколько городов и стран, а затем нажмите кнопку Submit для отправки формы на сервер. Механизм связывания найдет и обработает значения данных с индексами, и будет использовать их для создания коллекции объектов AddressSummary, которая затем будет передана обратно в представление и выведена на экран, как показано на рисунке 22-9.

**Рисунок 22-9:** Связывание коллекций пользовательских объектов



# ВЫЗОВ МОДЕЛИ СВЯЗЫВАНИЯ ДАННЫХ ВРУЧНУЮ

Если метод действия определяет параметры, процесс связывания данных будет запущен автоматически, но мы можем также взять контроль над ним в свои руки. Таким образом, мы получим более явный контроль над тем, как создаются экземпляры объектов моделей, откуда поступают значения данных, и как обрабатываются ошибки анализа данных. В листинге 22-28 показано, как мы изменили наш метод действия Address в контроллере Home, чтобы вручную запустить процесс связывания.

**Листинг 22-28:** Запускаем процесс связывания данных вручную в методе действия Address

```
public ActionResult Address()
{
    IList<AddressSummary> addresses = new List<AddressSummary>();
    UpdateModel(addresses);
    return View(addresses);
}
```

Метод UpdateModel принимает объект модели, который мы ранее определили в качестве его параметра, и пытается извлечь значения из его открытых свойств, запуская стандартный процесс связывания.

Когда мы вызываем процесс связывания вручную, мы можем ограничить его одним источником данных. По умолчанию механизм связывания проверяет четыре адреса – данные формы, данные маршрута, строку запроса и любые загруженные файлы. В листинге 22-29 показано, как мы можем ограничить поиск механизма связывания одним адресом, в этом случае, данными формы.

**Листинг 22-29:** Ограничиваем поиск механизма связывания данными формы

```
public ActionResult Address()
{
    IList<AddressSummary> addresses = new List<AddressSummary>();
    UpdateModel(addresses, new FormValueProvider(ControllerContext));
    return View(addresses);
}
```

Эта версия метода UpdateModel принимает реализацию интерфейса IValueProvider, который становится единственным источником значений данных для процесса связывания. Для каждого из четырех адресов данных по умолчанию есть реализация IValueProvider, как показано в таблице 22-2.

**Таблица 22-2:** Встроенные реализации IValueProvider

Адрес	Реализация IValueProvider
Request.Form	FormValueProvider
RouteData.Values	RouteDataProvider
Request.QueryString	QueryStringValueProvider
Request.Files	HttpFileCollectionValueProvider

Каждый из классов, перечисленных в таблице 22-2, принимает параметр конструктора ControllerContext, который мы получаем через свойство ControllerContext, определенное в классе Controller, как показано в листинге.

Наиболее распространенный способ ограничить источники данных – это проверять только значения формы. Существует удобная техника связывания, которая не требует создавать экземпляр FormValueProvider, как показано в листинге 22-30.

### Листинг 22-30: Ограничаем источники данных для механизма связывания

```
public ActionResult Address (FormCollection formData)
{
    IList<AddressSummary> addresses = new List<AddressSummary>();
    UpdateModel(addresses, formData);
    return View(addresses);
}
```

Класс `FormCollection` реализует интерфейс `IValueProvider`, и если мы определяем метод действия, который принимает параметр этого типа, механизм связывания создаст объект, который мы сможем передать непосредственно в метод `UpdateModel`.

#### Подсказка

*Есть и другие перегруженные версии метода `UpdateModel`, которые позволяют указать префикс для поиска и свойства модели, которые должны быть включены в процесс связывания.*

## Работаем с ошибками связывания

Пользователи неизбежно будут отправлять значения, которые не могут быть связаны с соответствующими свойствами модели – например, недействительные даты или текст для числовых значений. При явном вызове процесса связывания мы берем на себя ответственность за обработку таких ошибок. Механизм связывания генерирует для ошибок связывания исключение `InvalidOperationException`. Подробности ошибки можно узнать с помощью функции `ModelState`, которую мы опишем в главе 23. Но при использовании метода `UpdateModel`, мы должны быть готовы поймать исключение и создать сообщение об ошибке для пользователя с помощью `ModelState`, как показано в листинге 22-31.

### Листинг 22-31: Работаем с ошибками связывания данных

```
public ActionResult Address (FormCollection formData)
{
    IList<AddressSummary> addresses = new List<AddressSummary>();
    try
    {
        UpdateModel(addresses, formData);
    }
    catch (InvalidOperationException ex)
    {
        // provide feedback to user
    }
    return View(addresses);
}
```

В качестве альтернативного подхода, можно использовать метод `TryUpdateModel`, который возвращает `true`, если процесс связывания прошел успешно, и `false`, если возникли ошибки, как показано в листинге 22-32.

### Листинг 22-32: Используем метод TryUpdateModel

```
public ActionResult Address (FormCollection formData)
{
    IList<AddressSummary> addresses = new List<AddressSummary>();
    if (TryUpdateModel(addresses, formData))
    {
        // proceed as normal
    }
```

```
        else
    {
        // provide feedback to user
    }
    return View(addresses);
}
```

Единственное преимущество метода TryUpdateModel над UpdateModel в том, что вам не нужно будет ловить исключения и обрабатывать их. Функциональных различий в процессе связывания нет.

#### Подсказка

*Когда связывание данных запускается автоматически, при возникновении ошибок исключения не генерируются. Мы должны проверять результат через свойство ModelState.IsValid. Мы рассмотрим ModelState в главе 23.*

## Настройка системы модели связывания данных

Мы рассмотрели процесс связывания данных по умолчанию. Как и следовало ожидать, в следующих разделах мы продемонстрируем вам несколько примеров настройки системы связывания.

### Создаем пользовательский провайдер значений

Определяя пользовательский провайдер значений, мы можем добавить в процесс связывания собственный источник данных. Провайдеры значений реализуют интерфейс IValueProvider, который показан в листинге 22-33.

#### Листинг 22-33: Интерфейс IValueProvider

```
namespace System.Web.Mvc
{
    public interface IValueProvider
    {
        bool ContainsPrefix(string prefix);
        ValueProviderResult GetValue(string key);
    }
}
```

Механизм связывания вызывает метод ContainsPrefix, чтобы определить, может ли провайдер значений предоставить данные для определенного префикса. Метод GetValue возвращает значение для данного ключа данных или null, если провайдер не имеет соответствующих данных.

Мы добавили в пример проекта папку Infrastructure и создали новый файл под названием CountryValueProvider.cs, с помощью которого будем предоставлять значения для свойства Country. Содержимое этого файла показано в листинге 22-34.

#### Листинг 22-34: Содержимое файла CountryValueProvider.cs

```
using System.Globalization;
using System.Web.Mvc;

namespace MvcModels.Infrastructure
{
    public class CountryValueProvider : IValueProvider
```

```

{
    public bool ContainsPrefix(string prefix)
    {
        return prefix.ToLower().IndexOf("country") > -1;
    }

    public ValueProviderResult GetValue(string key)
    {
        if (ContainsPrefix(key))
        {
            return new ValueProviderResult("USA", "USA",
                CultureInfo.InvariantCulture);
        }
        else
        {
            return null;
        }
    }
}
}

```

Этот провайдер значений отвечает только на запросы значений для свойства `Country`, и он всегда возвращает значение `USA`. Для всех других запросов мы возвращаем `null`, указывая, что не можем предоставить данные.

Мы должны возвращать значения данных как класс `ValueProviderResult`. Этот класс имеет три параметра конструктора. Первый - это данные, которые мы хотим связать с запрошенным ключом. Второй параметр представляет собой версию значения данных, которая является безопасной для отображения. Последний параметр - это информация о культуре, которая относится к значению; мы указали `InvariantCulture`.

Чтобы зарегистрировать провайдер значений в приложении, нам нужно создать фабрику, которая будет создавать экземпляры нашего провайдера. Этот класс можно наследовать от абстрактного класса `ValueProviderFactory`. В листинге 22-35 показано содержимое файла `CustomValueProviderFactory.cs`, который мы создали в папке `Infrastructure`.

### Листинг 22-35: Содержимое файла `CustomValueProviderFactory.cs`

```

using System.Web.Mvc;

namespace MvcModels.Infrastructure
{
    public class CustomValueProviderFactory : ValueProviderFactory
    {
        public override IValueProvider GetValueProvider(ControllerContext
controllerContext)
        {
            return new CountryValueProvider();
        }
    }
}

```

Когда механизму связывания необходимо получить значения для процесса связывания, он вызывает метод `GetValueProvider`. Наша реализация просто создает и возвращает экземпляр класса `CountryValueProvider`, но можно использовать данные, предоставленные через параметр `ControllerContext`, чтобы реагировать на разные запросы и создавать для них разные провайдеры значений.

Зарегистрировать класс фабрики в приложении можно в методе `Application_Start` файла `Global.asax`, как показано в листинге 22-36.

**Листинг 22-36:** Регистрируем фабрику провайдеров значений

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Http;
using System.Web.Mvc;
using System.Web.Optimization;
using System.Web.Routing;
using MvcModels.Infrastructure;

namespace MvcModels
{
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();

            ValueProviderFactories.Factories.Insert(0, new CustomValueProviderFactory());

            WebApiConfig.Register(GlobalConfiguration.Configuration);
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);
        }
    }
}
```

Чтобы зарегистрировать класс фабрики, мы добавляем экземпляр в статическую коллекцию `ValueProviderFactories.Factories`. Механизм связывания просматривает провайдеры значений по очереди, следовательно, если мы хотим иметь приоритет надстроенными провайдерами, то можем поставить наш пользовательский провайдер на первое место в коллекции с помощью метода `Insert`.

Если мы хотим сделать наш провайдер резервным и использовать его только тогда, когда другие провайдеры не могут предоставить значение данных, то добавим его в конец коллекции с помощью метода `Add`:

```
ValueProviderFactories.Factories.Add(new CustomValueProviderFactory());
```

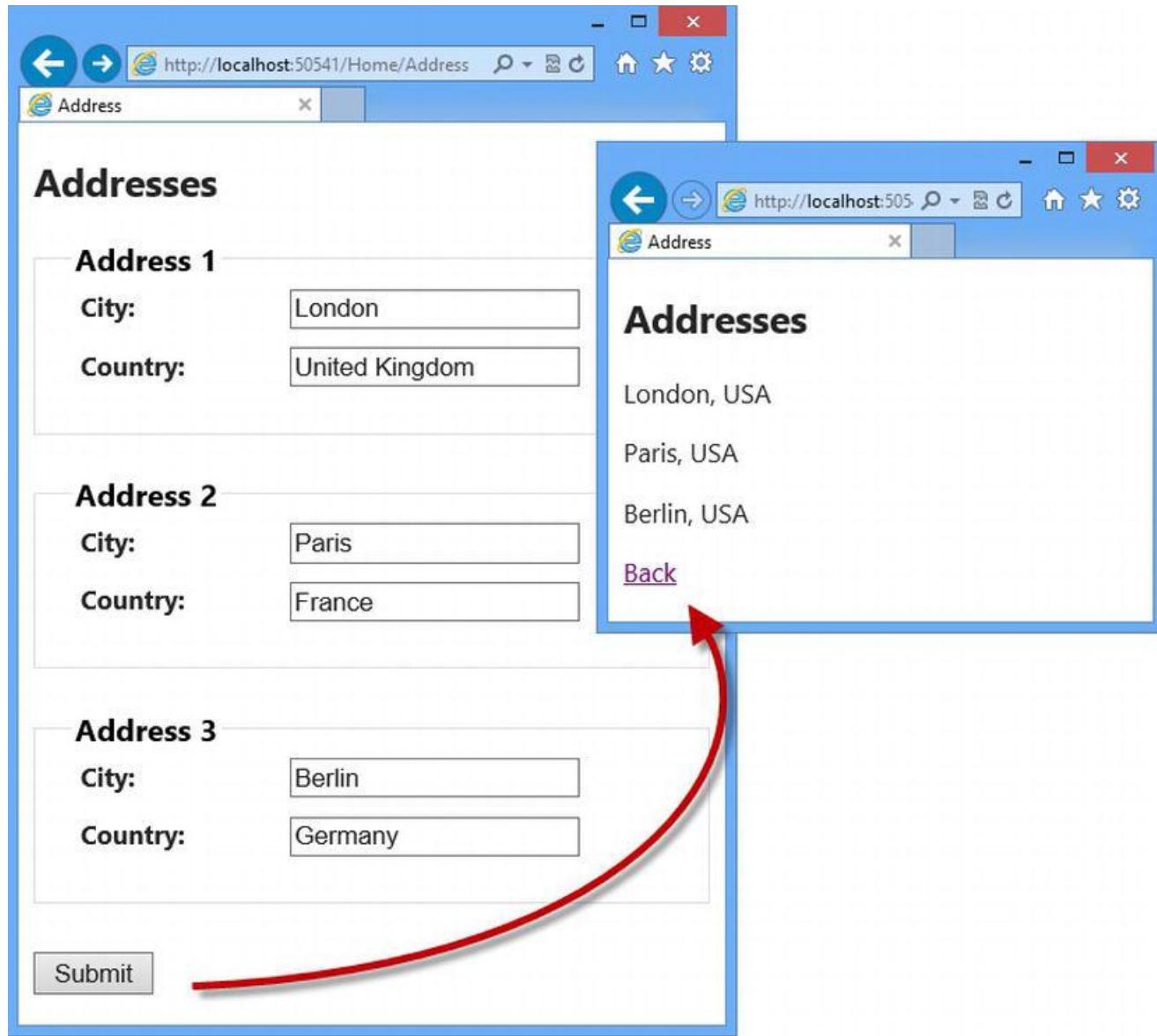
В данном примере мы хотим, чтобы наш провайдер значений использовался прежде любого другого провайдера, и поэтому мы добавили его с помощью метода `Insert`. Прежде чем мы сможем протестировать наш провайдер значений, необходимо изменить метод действия `Address`, чтобы механизм связывания проверял не только данные формы для поиска значений для свойств модели. В листинге 22-37 показано, как мы сняли ограничение на источник значений в методе `UpdateModel`.

**Листинг 22-37:** Снимаем ограничения на источники значений для свойств модели

```
public ActionResult Address()
{
    IList<AddressSummary> addresses = new List<AddressSummary>();
    UpdateModel(addresses);
    return View(addresses);
}
```

Чтобы увидеть работу нашего пользовательского провайдера, запустите приложение и перейдите по ссылке /Home/Address. Введите данные в поля city и Country и нажмите кнопку Submit. Вы увидите, что наш пользовательский провайдер значений, который имеет приоритет над встроенным провайдером, сгенерировал значения для свойства Country каждого объекта AddressSummary, который был создан механизмом связывания, как показано на рисунке 22-10.

**Рисунок 22-10:** Эффект пользовательского провайдера значений



### Создаем пользовательский механизм связывания

Чтобы изменить поведение механизма связывания по умолчанию, можно создать пользовательской механизм связывания для определенного типа.

Пользовательский механизм связывания реализует интерфейс `IModelBinder`, который мы показали вам ранее в этой главе. Чтобы его продемонстрировать, мы добавили в папку `Infrastructure` файл `AddressSummaryBinder.cs`, содержимое которого вы можете увидеть в листинге 22-38.

**Листинг 22-38:** Содержимое класса `AddressSummaryBinder.cs`

```
using MvcModels.Models;
```

```

using System.Web.Mvc;

namespace MvcModels.Infrastructure
{
    public class AddressSummaryBinder : IModelBinder
    {
        public object BindModel(ControllerContext controllerContext,
                               ModelBindingContext bindingContext)
        {
            AddressSummary model = (AddressSummary) bindingContext.Model
                ?? new AddressSummary();
            model.City = GetValue(bindingContext, "City");
            model.Country = GetValue(bindingContext, "Country");
            return model;
        }

        private string GetValue(ModelBindingContext context, string name)
        {
            name = (context.ModelName == "" ? "" : context.ModelName + ".") + name;
            ValueProviderResult result = context.ValueProvider.GetValue(name);
            if (result == null || result.AttemptedValue == "")
            {
                return "<Not Specified>";
            }
            else
            {
                return (string) result.AttemptedValue;
            }
        }
    }
}

```

Чтобы создать экземпляр типа модели, который поддерживается механизмом связывания, MVC Framework вызовет метод `BindModel`. Мы зарегистрируем механизм связывания немного позже. Наш класс `AddressSummaryBinder` будет создавать только экземпляры класса `AddressSummary`, что намного упростит код (можно создавать пользовательские механизмы связывания, которые поддерживают несколько типов, но мы предпочитаем один механизм для одного типа).

#### **Подсказка**

*В этом механизме связывания мы не выполняем валидацию входных данных, беспечно предполагая, что пользователь предоставит допустимые значения для всех свойств объекта Person. Мы рассмотрим валидацию в главе 23, а здесь сосредоточимся на базовом процессе связывания данных.*

Параметрами метода `BindModel` являются объект `ControllerContext`, с помощью которого можно получить информацию о текущем запросе, и объект `ModelBindingContext`, который предоставляет подробную информацию об искомом объекте модели, а также доступ к остальным возможностям связывания данных в приложении MVC. В таблице 22-3 описаны наиболее полезные свойства, определенные классом `ModelBindingContext`.

**Таблица 22-3:** Наиболее полезные свойства класса `ModelBindingContext`

Свойство	Описание
<code>Model</code>	Возвращает объект модели, переданный в метод <code>UpdateModel</code> , если связывание было вызвано вручную.
<code>ModelName</code>	Возвращает имя модели, которая участвует в процессе связывания.

Свойство	Описание
ModelType	Возвращает тип создаваемой модели.
ValueProvider	Возвращает реализацию <code>IValueProvider</code> , с помощью которой можно получить значения данных из запроса.

Наш пользовательский механизм связывания очень прост. Получив вызов метода `BindModel`, мы проверяем, было ли установлено свойство `Model` объекта `ModelBindingContext`. Если это так, то мы будем генерировать значение данных для этого объекта, а если нет, то мы создадим новый экземпляр класса `AddressSummary`. Чтобы получить значения для свойств `City` и `Country`, мы вызываем метод `GetValue`, а потом возвращаем заполненный объект `AddressSummary`.

В методе `GetValue` мы используем реализацию `IValueProvider`, полученную из свойства `ModelBindingContext.ValueProvider`, чтобы получить значения для свойств объекта модели.

Свойство `ModelName` сообщает нам, нужно ли добавить к имени искомого свойства какой-либо префикс. Если помните, наш метод действия пытается создать коллекцию объектов `AddressSummary`, следовательно, отдельные элементы `input` будет иметь значения атрибутов `name` с префиксами `[0]` и `[1]`. В запросе мы будем искать значения `[0].City`, `[0].Country` и так далее. Наконец, мы поставляем значение по умолчанию `<Not Specified>`, если не можем найти значение для свойства или свойство является пустой строкой (то есть пользователь не ввел значение в элемент `input` и отправил форму на сервер).

## Регистрируем пользовательский механизм связывания

Мы должны зарегистрировать наш механизм связывания, чтобы сообщить приложению, какие типы он поддерживает. Это можно сделать в методе `Application_Start` файла `Global.asax`, как показано в листинге 22-39.

**Листинг 22-39:** Регистрируем пользовательский механизм связывания

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Http;
using System.Web.Mvc;
using System.Web.Optimization;
using System.Web.Routing;
using MvcModels.Infrastructure;
using MvcModels.Models;

namespace MvcModels
{
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();

            // This statement has been commented out
            //ValueProviderFactories.Factories.Insert(0,
            // new CustomValueProviderFactory());

            ModelBinders.Binders.Add(typeof(AddressSummary), new AddressSummaryBinder());

            WebApiConfig.Register(GlobalConfiguration.Configuration);
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
        }
    }
}
```

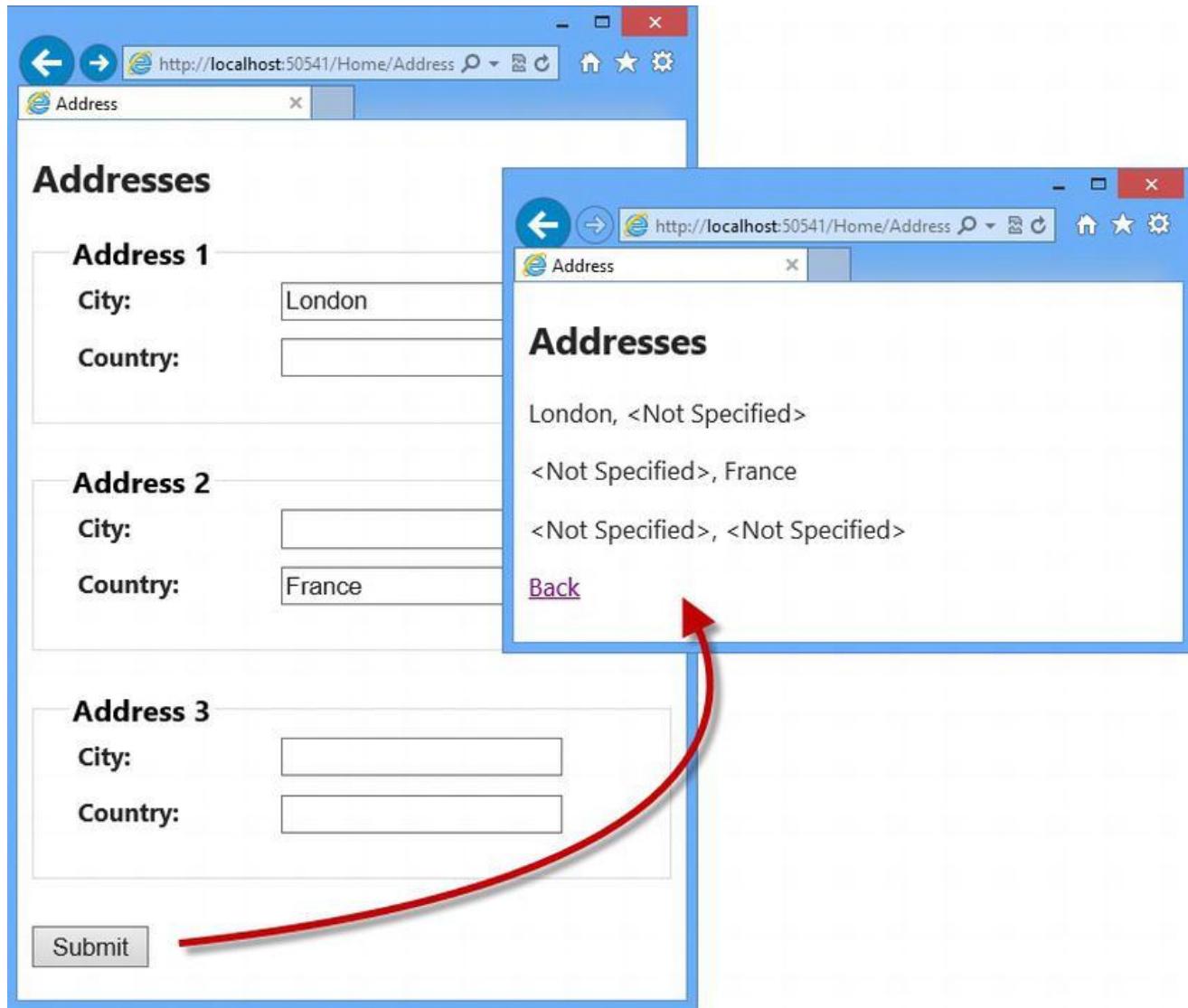
```

        BundleConfig.RegisterBundles(BundleTable.Bundles);
    }
}
}

```

Мы регистрируем наш механизм связывания с помощью метода `ModelBinders.Binders.Add`, передавая в него тип, который поддерживает наш механизм связывания, и экземпляр класса механизма связывания. Обратите внимание, что мы удалили оператор, который регистрирует пользовательский провайдер значений. Чтобы проверить работу пользовательского механизма связывания, запустите приложение, перейдите по ссылке `/Home/Address` и заполните несколько элементов формы. Когда вы отправите форму, наш механизм связывания будет использовать `<Not Specified>` для всех свойств, для которых вы не ввели значений, как показано на рисунке 22-11.

**Рисунок 22-11:** Эффект использования пользовательского механизма связывания



#### Подсказка

*Как вариант, вы можете задать пользовательские механизмы связывания, применив атрибут `ModelBinder` к классу модели.*

## Резюме

В этой главе мы продемонстрировали вам процесс связывания данных, показали работу механизма связывания по умолчанию и различные способы настройки процесса. Для многих приложений MVC Framework потребуется только механизм связывания по умолчанию; он хорошо работает с HTML, который генерируют вспомогательные методы. Но для более сложных приложений может пригодиться пользовательский механизм связывания, который будет создавать объекты моделей более эффективным для определенных ситуаций способом. В следующей главе мы покажем вам, как проводить валидацию объектов моделей и как отображать пользователю ошибки, возникающие при вводе недействительных данных.

# Валидация модели

В предыдущей главе мы рассмотрели, как MVC Framework создает объекты моделей из запросов HTTP с помощью механизма связывания данных. Тогда мы предполагали, что данные, предоставленные пользователями, являются действительными. На самом деле, пользователи будут часто вводить данные, с которыми мы не сможем работать, что подводит нас к теме этой главы - валидация моделей.

Валидация моделей - это подтверждение того, что полученные данные пригодны для связывания с моделью, и, если это не так, отображение пользователю сообщения с описанием проблемы.

Первая часть процесса - проверка полученных данных - позволяет сохранить целостность доменной модели. Фильтруя данные, которые не имеют смысла в контексте нашего домена, мы предотвращаем возникновение странных и нежелательных состояний в нашем приложении. Вторая часть - помочь пользователю в исправлении ошибки - не менее важна. Если мы не предоставим пользователю информацию и инструменты, необходимые для нормального взаимодействия с нашим приложением, то сбьем их с толку и вызовем их недовольство. Если это общедоступное приложение, то пользователи просто перестанут его использовать. Если это корпоративное приложение, то рабочий процесс будет затруднен. Ни то, ни другое для нас не желательно. К счастью, MVC Framework предоставляет расширенную поддержку валидации моделей. Мы рассмотрим, как использовать ее базовые функции, а затем продемонстрируем продвинутые техники для тонкой настройки процесса валидации.

## Создание проекта для примера

Прежде чем начать, мы создадим простое приложение MVC, к которому будем применять различные техники валидации моделей. Мы создали новый проект MVC под названием `ModelValidation` на шаблоне `Basic`, а также новый класс модели под названием `Appointment.cs`, который показан в листинге 23-1.

**Листинг 23-1:** Класс модели `Appointment`

```
using System;
using System.ComponentModel.DataAnnotations;
namespace ModelValidation.Models
{
    public class Appointment
    {
        public string ClientName { get; set; }
        [DataType(DataType.Date)]
        public DateTime Date { get; set; }
        public bool TermsAccepted { get; set; }
    }
}
```

В классе модели `Appointment` мы определили три свойства и указали с помощью атрибута `DataType`, что свойство `Date` должно быть выражено в формате даты без компонента времени.

Для этого примера мы также создали контроллер `Home` и определили методы действий, которые работают с классом модели `Appointment`, как показано в листинге 23-2.

**Листинг 23-2:** Контроллер `Home` в проекте `ModelValidation`

```
using System;
using System.Web.Mvc;
```

```

using ModelValidation.Models;

namespace ModelValidation.Controllers
{
    public class HomeController : Controller
    {
        public ViewResult MakeBooking()
        {
            return View(new Appointment { Date = DateTime.Now });
        }

        [HttpPost]
        public ViewResult MakeBooking(Appointment appt)
        {
            // statements to store new Appointment in a
            // repository would go here in a real project
            return View("Completed", appt);
        }
    }
}

```

Как и в предыдущих главах, мы определили две версии метода действия `MakeBooking`. В этой главе нас интересует версия с атрибутом `HttpPost`, так как именно в ней будет применяться связывание данных для создания объекта параметра `Appointment`.

Обратите внимание, что в реальном приложении мы указали бы в комментариях, где находятся операторы для сохранения информации об объекте `Appointment`, который будет создан механизмом связывания. В этом примере мы не собираемся создавать хранилище, потому что хотим сосредоточиться на процессах связывания данных и валидации. Однако, важно помнить, что цель валидации модели - предотвратить сохранение в хранилище нежелательных или бессмысленных данных и возникновение проблем (при попытке их сохранения или, в дальнейшем, обработки).

Чтобы завершить наш пример приложения, создадим пару представлений для работы с методами действий в папке `/Views/Home`. В листинге 23-3 показано содержимое файла `MakeBooking.cshtml`, который содержит форму для создания новой записи на прием.

### Листинг 23-3: Содержимое файла `MakeBooking.cshtml`

```

@model ModelValidation.Models.Appointment

@{
    ViewBag.Title = "Make A Booking";
}

<h4>Book an Appointment</h4>

@using (Html.BeginForm())
{
    <p>Your name: @Html.EditorFor(m => m.ClientName)</p>
    <p>Appointment Date: @Html.EditorFor(m => m.Date)</p>
    <p>@Html.EditorFor(m => m.TermsAccepted) I accept the terms & conditions</p>
    <input type="submit" value="Make Booking" />
}

```

Когда форма будет отправлена обратно к приложению, метод действия `MakeBooking` отобразит подробную информацию о созданной пользователем записи с помощью представления `Completed.cshtml`, которое показано в листинге 23-4.

### Листинг 23-4: Содержимое файла `Completed.cshtml`

```
@model ModelValidation.Models.Appointment
```

```

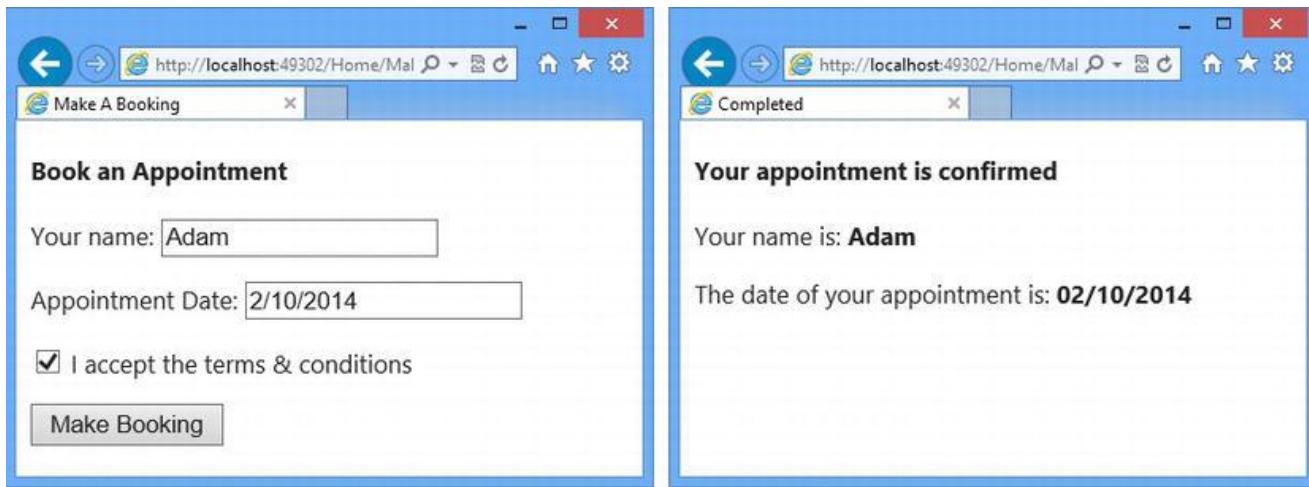
@{
    ViewBag.Title = "Completed";
}

<h4>Your appointment is confirmed</h4>
<p>Your name is: <b>@Html.DisplayFor(m => m.ClientName)</b></p>
<p>The date of your appointment is: <b>@Html.DisplayFor(m => m.Date)</b></p>

```

Как вы уже поняли, наш пример для этой главы основан на создании записей на прием. Чтобы увидеть, как это работает, запустите приложение и перейдите по ссылке `Home/MakeBooking`. Введите реквизиты в форму и нажмите кнопку `Submit`, чтобы отправить данные на сервер; он выполнит связывание данных и создаст объект `Appointment`, информация о котором затем будет отображена в представлении `Completed.cshtml`, как показано на рисунке 23 -1.

**Рисунок 23-1:** Используем пример приложения



Сейчас наше приложение будет принимать любые данные, которые отправит пользователь, но, чтобы сохранить целостность нашего приложения и доменной модели, мы будем требовать выполнения трех условий, прежде чем принять отправленный пользователем объект `Appointment`:

- Пользователь должен указать имя.
- Пользователь должен указать дату (в формате `mm/dd/yyyy`), которая относится к будущему.
- Пользователь должен отметить чекбокс, чтобы принять условия пользовательского соглашения.

Валидация модели – это процесс проверки соблюдения этих требований. В следующих разделах мы покажем вам различные техники для проверки указанных пользователем данных и предоставления обратной связи в тех случаях, когда мы не можем использовать полученные данные.

## Явная валидация модели

Наиболее очевидный способ валидации модели – проверить данные в соответствующем методе действия. В листинге 23-5 показано, как мы добавили явную проверку для каждого свойства, определенного в классе `Appointment`, в `HttpPost`-версии метода действия `MakeBooking`.

**Листинг 23-5:** Осуществляем явную валидацию модели

```

[HttpPost]
public ViewResult MakeBooking(Appointment appt)

```

```

{
    if (string.IsNullOrEmpty(appt.ClientName))
    {
        ModelState.AddModelError("ClientName", "Please enter your name");
    }

    if (ModelState.IsValidField("Date") && DateTime.Now > appt.Date)
    {
        ModelState.AddModelError("Date", "Please enter a date in the future");
    }

    if (!appt.TermsAccepted)
    {
        ModelState.AddModelError("TermsAccepted", "You must accept the terms");
    }

    if (ModelState.IsValid)
    {
        // statements to store new Appointment in a
        // repository would go here in a real project
        return View("Completed", appt);
    }
    else
    {
        return View();
    }
}

```

Мы проверяем значения, которые механизм связывания присвоил свойствам объекта параметра, и регистрируем ошибки, которые находим в свойстве `ModelState` (его наш контроллер наследует от базового класса). Например, свойство `ClientName` мы проверяем таким образом:

```

if (string.IsNullOrEmpty(appt.ClientName)) {
    ModelState.AddModelError("ClientName", "Please enter your name");
}

```

Мы хотим, чтобы пользователь ввел значение для этого свойства, поэтому используем статический метод `string.IsNullOrEmpty`, чтобы его проверить. Если мы не получили значение, то с помощью метода `ModelState.AddModelError` указываем имя свойства, с которым возникла проблема (`ClientName`), и сообщение, которое будет показано пользователю, чтобы помочь ему исправить проблему (`Please enter your name`).

С помощью свойства `ModelState.IsValidField` мы можем проверить, смог ли механизм связывания присвоить значение. Мы применяем его в свойстве `Date`, чтобы убедиться, что механизм связывания смог проанализировать значение, полученное от пользователя; нет никакого смысла выполнять дополнительные проверки или регистрировать дополнительные ошибки, если значение не могло быть выделено из данных запроса.

После того как мы провели валидацию всех свойств в объекте модели, мы проверяем свойство `ModelState.IsValid` на наличие ошибок. Если во время проверок мы вызвали метод `ModelState.AddModelError`, или у механизма связывания возникли проблемы с созданием объекта `Appointment`, то этот метод возвращает `true`.

```

if (ModelState.IsValid)
{
    // statements to store new Appointment in a
    // repository would go here in a real project
    return View("Completed", appt);
}
else

```

```
{  
    return View();  
}
```

Если в свойстве `IsValid` нет ошибок, то наш объект `Appointment` является действительным, и мы можем визуализировать представление `Completed.cshtml` (и, в реальном проекте, сохранить объект `Appointment` в хранилище). Если свойство `IsValid` возвращает `false`, то мы знаем, что у нас есть проблема, и вызываем метод `View`, который визуализирует представление по умолчанию.

## Отображаем ошибки валидации пользователю

Может показаться странным, что мы обрабатываем ошибки валидации с помощью метода `View`, но шаблонные вспомогательные методы, которые мы использовали для создания элементов ввода в представлении `MakeBooking.cshtml`, проверяют модель представления на наличие ошибок валидации.

Если в свойстве появилась ошибка, вспомогательный метод добавляет к элементу ввода класс CSS под названием `input-validation-error`. Файл `~/Content/Site.css` содержит определение по умолчанию для этого стиля, которое выглядит следующим образом:

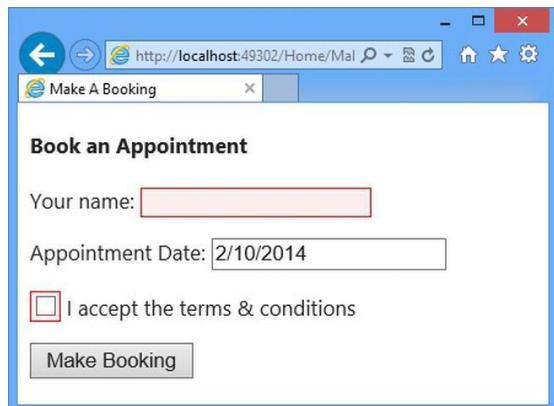
```
.input-validation-error {  
    border: 1px solid #f00;  
    background-color: #fee;  
}
```

Он устанавливает красную рамку и розовый фон для элемента, в котором есть ошибка. Чтобы проверить работу явной валидации, запустите приложение, перейдите по ссылке `/Home/MakeBooking` и нажмите кнопку `Make Booking`, не вводя данные в форму. Результат показан на рисунке 23-2.

### Подсказка

*Если вы хотите написать собственные вспомогательные методы, которые поддерживают валидацию, изучите исходный код класса `System.Mvc.Web.Html.InputExtensions`, чтобы понять, как это сделаеть. Вкратце скажем, что класс `System.Web.Mvc.HtmlHelper` определяет метод `GetModelStateValue`, который позволяет проверить, есть ли ошибки валидации для конкретного свойства. Есть и дополнительные методы, которые позволяют легко получить необходимый CSS и сообщения об ошибках для создания предупреждения пользователю.*

Рисунок 23-2: Поля с ошибками подсвечены



## Применяем стили к чекбоксам

Некоторые браузеры, в том числе Chrome и Firefox, игнорируют стили, примененные к чекбоксам, в результате чего элементы формы будут выглядеть непоследовательно. Чтобы решить эту проблему, можно заменить шаблон Boolean для элементов editor пользовательским шаблоном ~/Views/Shared/EditorTemplates/Boolean.cshtml и заключить чекбокс в элемент div. Мы используем следующий шаблон, который вы можете приспособить для своего приложения:

Этот шаблон поместит чекбокс в элемент div, к которому будет применен стиль input-validation-error, если возникнут какие-либо ошибки модели, связанные с тем свойством, к которому был применен шаблон. Вы можете узнать больше о замене шаблонов editor в главе 20.

Если вы отправите форму без данных, ошибки будут подсвечены для свойств ClientName и TermsAccepted. Значение по умолчанию для свойства Date является действительным, поэтому ошибка валидации не возникает.

Пользователь не увидит представление Completed.cshtml, пока не отправит форму с данными, которые могут быть проанализированы браузером модели (model browser), который передает операторы явной валидации в метод действия MakeBooking. В противном случае отправка формы будет визуализирована представление MakeBooking.cshtml с текущими ошибками валидации.

## Отображение сообщений валидации

CSS стили, которые применяются шаблонными вспомогательными методами к элементам ввода, указывают на проблему с полем, но не сообщают пользователю, в чем эта проблема заключается. К счастью, для этого существуют другие вспомогательные методы. В листинге 23-6 показан один из таких методов, который мы применили к представлению MakeBooking (так как именно оно демонстрирует ошибки валидации пользователю).

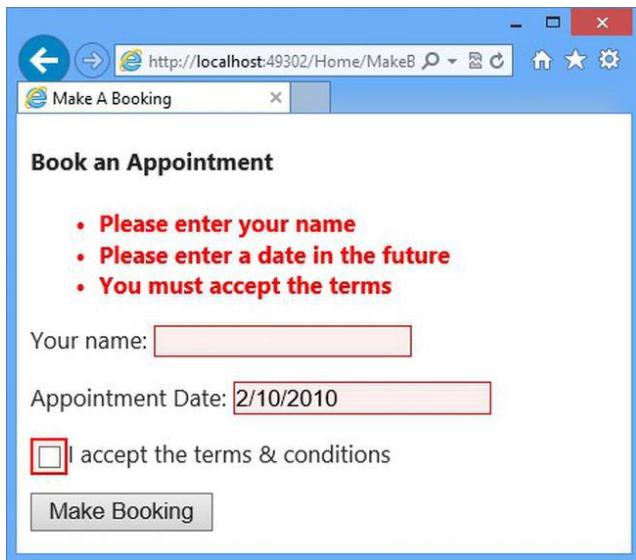
**Листинг 23-6:** Используем вспомогательный метод ValidationSummary

```
<h4>Book an Appointment</h4>

@using (Html.BeginForm())
{
    @Html.ValidationSummary()
    <p>Your name: @Html.EditorFor(m => m.ClientName)</p>
    <p>Appointment Date: @Html.EditorFor(m => m.Date)</p>
    <p>@Html.EditorFor(m => m.TermsAccepted) I accept the terms & conditions</p>
    <input type="submit" value="Make Booking" />
}
```

Вспомогательный метод Html.ValidationSummary добавляет сообщение об ошибках валидации, которые были зарегистрированы на отображаемой пользователю странице. Если ошибок нет, то вспомогательный метод не генерирует HTML. На рисунке 23-3 показан результат применения этого метода. Мы очистили поля ввода и отправили форму.

**Рисунок 23-3:** Отображение сообщения валидации



#### Примечание

Для свойства `Date` мы используем значения в американском формате даты `month/day/year`. Если вы находитесь в другой стране, то можете ввести допустимые даты в местном формате (например, `day/month/year`, который широко используется в Европе) или добавить для этого проекта строку `<globalization culture="en-US" uiCulture="en-US"/>` в элемент `system.web` в файле `Web.config`, чтобы приложение использовало формат даты US.

В сообщении валидации отображается информация об ошибках, которые были зарегистрированы в `ModelState` в методе действия `MakeBooking`. Этот вспомогательный метод генерирует следующий HTML:

```
<div class="validation-summary-errors" data-valmsg-summary="true">
  <ul>
    <li>Please enter your name</li>
    <li>Please enter a date in the future</li>
    <li>You must accept the terms</li>
  </ul>
</div>
```

Ошибки выводятся в виде списка в элементе `div`, к которому применен стиль `validation-summary-errors`. Этот стиль определен в `~/Content/Site.css` и по мере необходимости может быть изменен. По умолчанию стиль выделяет текст сообщений об ошибках полужирным и красным:

```
.validation-summary-errors {
  font-weight: bold;
  color: #f00;
}
```

Для метода `ValidationSummary` существует несколько перегруженных версий; наиболее полезные из них показаны в таблице 23-1. Некоторые из этих версий позволяют указать, что нужно отображать только ошибки уровня модели (*model-level errors*). Ошибки, которые до сих пор были зарегистрированы в `ModelState`, относятся к уровню свойства (*property-level errors*) и означают, что для данного свойства указано недействительное значение; если его изменить, проблема будет решена.

**Таблица 23-1:** Полезные перегруженные версии вспомогательного метода ValidationSummary

Перегруженный метод	Описание
Html.ValidationSummary()	Генерирует сообщения для всех ошибок валидации.
Html.ValidationSummary(bool)	Если параметр <code>bool</code> содержит <code>true</code> , то отображаются только ошибки уровня модели (см. пояснение после таблицы). Если параметр содержит <code>false</code> , то отображаются все ошибки.
Html.ValidationSummary(string)	Отображает сообщение (содержащееся в параметре <code>string</code> ) перед сообщениями обо всех ошибках валидации.
Html.ValidationSummary(bool, string)	Отображает сообщение перед сообщениями об ошибках валидации. Если параметр <code>bool</code> содержит <code>true</code> , то отображаются только ошибки уровня модели.

С другой стороны, ошибки уровня модели можно использовать, если проблемы возникают из-за взаимодействия двух и более значений свойств. Простой пример: давайте представим, что клиенту по имени Джо нельзя записываться на прием по понедельникам. В листинге 23-7 показано, как для этого правила можно создать явное условие валидации в методе действия MakeBooking и вывести сообщение об ошибке валидации уровня модели.

#### Листинг 23-7: Ошибка валидации уровня модели

```
[HttpPost]
public ViewResult MakeBooking(Appointment appt)
{
    if (string.IsNullOrEmpty(appt.ClientName))
    {
        ModelState.AddModelError("ClientName", "Please enter your name");
    }

    if (ModelState.IsValidField("Date") && DateTime.Now > appt.Date)
    {
        ModelState.AddModelError("Date", "Please enter a date in the future");
    }

    if (!appt.TermsAccepted)
    {
        ModelState.AddModelError("TermsAccepted", "You must accept the terms");
    }

    if (ModelState.IsValidField("ClientName") && ModelState.IsValidField("Date")
        && appt.ClientName == "Joe" && appt.Date.DayOfWeek == DayOfWeek.Monday)
    {
        ModelState.AddModelError("", "Joe cannot book appointments on Mondays");
    }

    if (ModelState.IsValid)
    {
        // statements to store new Appointment in a
        // repository would go here in a real project
        return View("Completed", appt);
    }
    else
    {
        return View();
    }
}
```

Прежде чем проверить, пытается ли Джо записаться на прием в понедельник, мы подтверждаем с помощью метода `ModelState.IsValidField`, что значения свойств `ClientName` и `Date` действительны.

Это означает, что мы не будем генерировать ошибку уровня модели, пока не проведем успешную валидацию предыдущих свойств. Чтобы зарегистрировать ошибку уровня модели, мы передаем пустую строку ("") в качестве первого параметра в метод `ModelState.AddModelError`, например:

```
ModelState.AddModelError("", "Joe cannot book appointments on Mondays");
```

Далее мы обновим файл `MakeBooking.cshtml`, чтобы использовать в нем версию вспомогательного метода `ValidationSummary`, которая принимает параметр `bool` и будет отображать только ошибки уровня модели, как показано в листинге 23-8.

**Листинг 23-8:** Обновляем представление `MakeBooking.cshtml`, чтобы отображать только ошибки уровня модели

```
@model ModelValidation.Models.Appointment

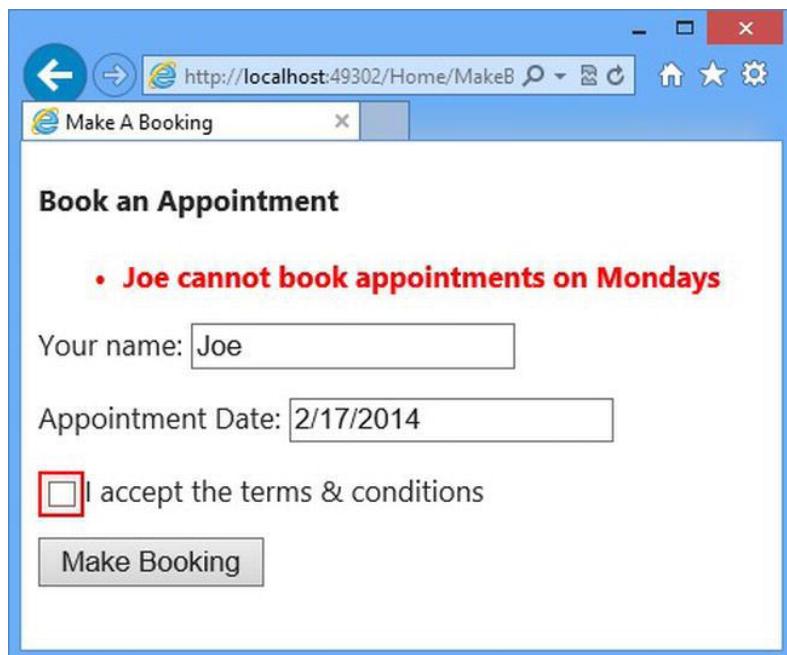
 @{
    ViewBag.Title = "Make A Booking";
}

<h4>Book an Appointment</h4>

@using (Html.BeginForm())
{
    @Html.ValidationSummary(true)
    <p>Your name: @Html.EditorFor(m => m.ClientName)</p>
    <p>Appointment Date: @Html.EditorFor(m => m.Date)</p>
    <p>@Html.EditorFor(m => m.TermsAccepted) I accept the terms & conditions</p>
    <input type="submit" value="Make Booking" />
}
```

Результат этих изменений показан на рисунке 23-4, где мы ввели имя Джо и указали дату, которая выпадает на понедельник.

**Рисунок 23-4:** Отображение сообщения валидации для ошибок уровня модели



Как видно из рисунка, есть две ошибки валидации. Первая - это ошибка уровня модели, которая возникает, когда Джо пытается записаться на прием в понедельник. Вторая появляется из-за того, что не отмечен чекбокс для условий пользовательского соглашения. Так как в сообщении валидации мы отображаем только ошибки уровня модели, пользователь не увидит никаких сведений о второй проблеме.

## Отображаем сообщения валидации уровня свойства

Мы отображаем только сообщения об ошибках уровня модели потому, что ошибки уровня свойства можно отображать рядом с соответствующими полями. Таким образом, мы не хотим дублировать сообщения, относящиеся к отдельным свойствам. В листинге 23-9 показано обновленное представление MakeBooking.cshtml, которое отображает ошибки уровня модели сверху и ошибки уровня свойства рядом с соответствующим полем ввода.

**Листинг 23-9:** Используем сообщения об ошибках для конкретного свойства

```
@model ModelValidation.Models.Appointment

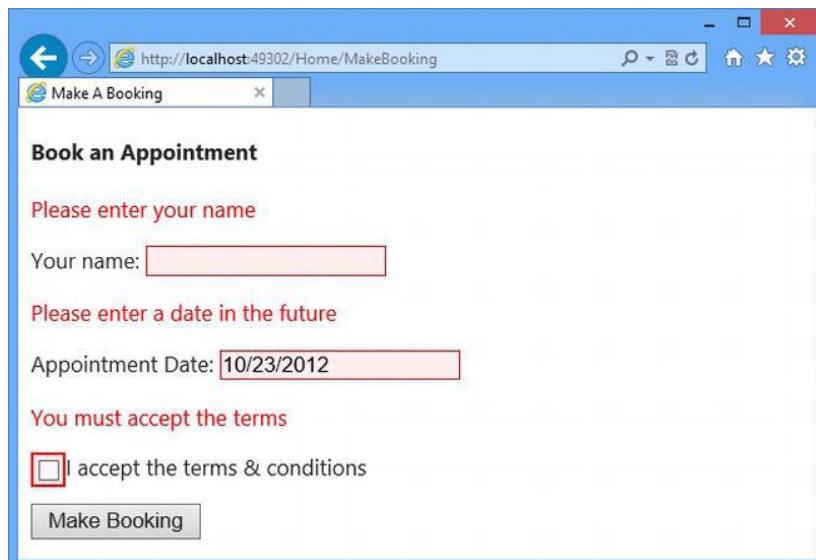
{@{
    ViewBag.Title = "Make A Booking";
}

<h4>Book an Appointment</h4>

@using (Html.BeginForm())
{
    @Html.ValidationSummary(true)
    <p>@Html.ValidationMessageFor(m => m.ClientName)</p>
    <p>Your name: @Html.EditorFor(m => m.ClientName)</p>
    <p>@Html.ValidationMessageFor(m => m.Date)</p>
    <p>Appointment Date: @Html.EditorFor(m => m.Date)</p>
    <p>@Html.ValidationMessageFor(m => m.TermsAccepted)</p>
    <p>@Html.EditorFor(m => m.TermsAccepted) I accept the terms & conditions</p>
    <input type="submit" value="Make Booking" />
}
```

Вспомогательный метод `Html.ValidationMessageFor` отображает ошибки валидации для одного свойства модели. Результат его применения в представлении MakeBooking показан на рисунке 23-5.

**Рисунок 23-5:** Используем вспомогательный метод для отображения сообщений валидации для одного свойства



The screenshot shows a web browser window with the URL <http://localhost:49302/Home/MakeBooking>. The page title is "Make A Booking". The main content is titled "Book an Appointment" and contains two red bullet points: "Joe cannot book appointments on Mondays" and "You must accept the terms". Below these are input fields for "Your name" (containing "Joe") and "Appointment Date" (containing "2/17/2014"). There is also a checkbox labeled "I accept the terms & conditions" which is unchecked. At the bottom is a "Make Booking" button.

## Выполнение валидации на стороне клиента

Техники валидации, которые мы продемонстрировали до сих пор, являются примерами валидации на стороне сервера. Это означает, что пользователь передает данные на сервер, сервер проверяет данные и отправляет обратно результат валидации (обработав данные, если валидация успешна, или отобразив список ошибок, которые должны быть исправлены).

В веб-приложениях пользователи обычно ожидают получить немедленную обратную связь перед тем, как они отправят данные на сервер. Это известно как валидация на стороне клиента и, как правило, реализуется с помощью JavaScript. Введенные данные проверяются перед отправкой на сервер, что обеспечивает пользователю немедленную обратную связь и возможность исправить все проблемы.

MVC Framework поддерживает *ненавязчивую валидацию на стороне клиента* (*unobtrusive client-side validation*). Термин «ненавязчивый» означает, что правила валидации выражаются с помощью атрибутов, добавленных к элементам HTML, которые мы создаем. Эти атрибуты интерпретируются библиотекой JavaScript, которая включена в MVC Framework; он, в свою очередь, использует библиотеку jQuery Validation, которая и выполняет всю работу по валидации. В следующих разделах мы покажем вам, как работает встроенная поддержка валидации и продемонстрируем способы расширения функциональности и создания пользовательской валидации на стороне клиента.

### Подсказка

*Валидация на стороне клиента ориентирована на проверку индивидуальных свойств. На самом деле, трудно создать валидацию уровня модели на стороне клиента с помощью встроенной поддержки MVC Framework. В связи с этим большинство приложений MVC используют валидацию на стороне клиента для уровня свойств и валидацию на стороне сервера для всей модели.*

## Активируем и dezактивируем валидацию на стороне клиента

Валидация на стороне клиента контролируется двумя настройками в файле Web.config, как показано в листинге 23-19.

### Листинг 23-19: Управляем валидацией на стороне клиента

```
<appSettings>
  <add key="ClientValidationEnabled" value="true"/>
  <add key="UnobtrusiveJavaScriptEnabled" value="true"/>
</appSettings>
```

Чтобы валидация на стороне клиента работала, оба эти параметра должны содержать значение true. При создании проекта MVC Visual Studio устанавливает для них значение true.

### Подсказка

Можно также настроить валидацию на стороне клиента для отдельных представлений, указав `HtmlHelper.ClientValidationEnabled` и `HtmlHelper.UnobtrusiveJavaScriptEnabled` в блоке кода Razor.

Для работы валидации на стороне клиента также необходимо, чтобы в коде HTML, отправленном в браузер, были ссылки на три библиотеки JavaScript:

- /Scripts/jquery-1.7.1.min.js
- /Scripts/jquery.validate.min.js
- /Scripts/jquery.validate.unobtrusive.min.js

Самый простой способ добавить эти JavaScript файлы в представление – с помощью *связок скриптов* (*script bundles*). Это новая функция в MVC 4, которую мы опишем в главе 24. Здесь мы не будем объяснять принципы ее работы, но добавим с ее помощью необходимые нам скрипты в файл /Views/Shared/\_Layout.cshtml; изменения показаны в листинге 23-20. (То же изменение можно было бы сделать в представлении MakeBooking, но мы предпочитаем импортировать скрипты в макеты, чтобы не приходилось вносить изменения во все представления).

### Листинг 23-20: Добавляем в макет библиотеки JavaScript, необходимые для валидации на стороне клиента

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width" />
  <title>@ViewBag.Title</title>
  @Styles.Render("~/Content/css")
  @Scripts.Render("~/bundles/modernizr")
</head>
<body>
  @RenderBody()
  @Scripts.Render("~/bundles/jquery")
  @Scripts.Render("~/bundles/jqueryval")
  @RenderSection("scripts", required: false)
</body>
</html>
```

## Используем валидацию на стороне клиента

Теперь, когда мы активировали валидацию на стороне клиента и убедились, что в макете есть ссылки на библиотеки JavaScript, можно выполнять валидацию на стороне клиента. Для этого проще всего применить атрибуты метаданных, которые мы использовали ранее для валидации на стороне сервера, такие как Required, Range и StringLength. В листинге 23-21 показан класс модели Appointment с этими атрибутами (мы удалили реализацию интерфейса IValidatableObject, который совершенно не влияет на валидацию на стороне клиента).

**Листинг 23-21:** Атрибуты валидации, примененные к объекту модели Appointment

```
using System;
using System.ComponentModel.DataAnnotations;

namespace ModelValidation.Models
{
    public class Appointment
    {
        [Required]
        [StringLength(10, MinimumLength = 3)]
        public string ClientName { get; set; }

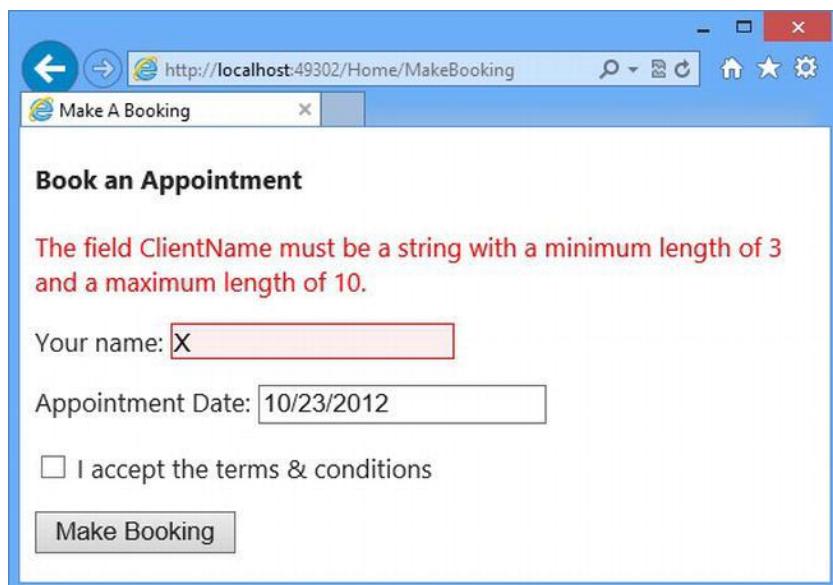
        [DataType(DataType.Date)]
        public DateTime Date { get; set; }

        public bool TermsAccepted { get; set; }
    }
}
```

Это все, что необходимо для базовой настройки валидации на стороне клиента. Мы применили несколько встроенных атрибутов валидации, чтобы продемонстрировать функции валидации на стороне клиента; если отправленный клиенту код HTML ссылается на библиотеки JavaScript, то все заработает.

Чтобы увидеть эффект от валидации на стороне клиента, запустите приложение, перейдите по ссылке /Home/MakeBooking и введите букву X в поле имени. Нажмите клавишу tab или кликните по другому элементу ввода, и вы сразу увидите сообщение валидации, созданное JavaScript, как показано на рисунке 23-11.

**Рисунок 23-11:** Немедленная обратная связь при использовании валидации на стороне клиента



В листинге 23-21 мы применили к классу Appointment атрибут валидации `StringLength`, и на рисунке вы увидите сообщение об ошибке от этого атрибута. Браузер обеспечивает немедленную обратную связь, не отправляя запросов к серверу. На самом деле, выполняющий валидацию код JavaScript будет препятствовать отправке формы до тех пор, пока все явные ошибки валидации не будут исправлены.

Исправляя ошибку, пользователь также получает немедленную обратную связь. Если вы вернетесь к полю `Name` и продолжите ввод, ошибка валидации исчезнет, когда в имени будет три и более символа. Но если вы продолжите печатать и дойдете до одиннадцатого символа, сообщение об ошибке появится снова. Это произойдет потому, что в атрибуте `StringLength` для свойства `ClientName` мы указали минимальную длину строки из трех букв, а максимальную – из десяти.

## Принципы работы валидации на стороне клиента

Одним из преимуществ использования валидации на стороне клиента в MVC Framework является то, что нам не придется писать JavaScript. Правила валидации создаются с помощью HTML-атрибутов. Когда валидация на стороне клиента отключена, вспомогательный метод `Html.EditorFor` создаст для свойства `ClientName` следующий код HTML:

```
<input class="text-box single-line" id="ClientName" name="ClientName" type="text" value="" />
```

А вот HTML, который будет создан для того же свойства при активированной валидации на стороне клиента:

```
<input class="text-box single-line" data-val="true" data-val-length="The field ClientName must be a string with a minimum length of 3 and a maximum length of 10." data-val-length-max="10" data-val-length-min="3" data-val-required="The ClientName field is required." id="ClientName" name="ClientName" type="text" value="" />
```

Поддержка валидации на стороне клиента в MVC не генерирует никакого JavaScript или данных JSON, которые бы управляли процессом валидации; как это часто бывает в работе с MVC Framework, здесь мы полагаемся на соглашения. Первым был добавлен атрибут `data-val`. С помощью этого атрибута библиотека jQuery Validation определяет те поля, для которых требуется валидация.

Индивидуальные правила валидации задаются с помощью атрибута в форме `data-val-<name>`, где `name` – это правило. Так, например, для атрибута `Required`, который мы применили к классу модели, был в HTML создан атрибут `data-val-required`. Значение в атрибуте – это сообщение об ошибке, связанное с правилом. Для некоторых правил требуются дополнительные атрибуты, как, например, для правила `length`, для которого атрибуты `data-val-length-min` и `data-val-length-max` позволяют задавать минимальную и максимальную длину строки.

Правила валидации `required` и `length` интерпретируются библиотекой JQuery Validation, на которой построены все функции валидации на стороне клиента MVC. Одна из приятных особенностей валидации на стороне клиента MVC заключается в том, что для создания правил валидации на стороне клиента и на сервере применяются одни и те же атрибуты. Это означает, что данные из браузеров, которые не поддерживают JavaScript, подлежат той же проверке, как и данные из браузеров, поддерживающих JavaScript, и это не требует от нас каких-либо дополнительных усилий.

## Валидация на стороне клиента MVC и валидация jQuery

Функции валидации на стороне клиента MVC построены на библиотеке JQuery Validation. Если хотите, можете использовать библиотеку Validation напрямую, не обращая внимания на функции MVC. Библиотека Validation очень гибкая и функционально богатая; ее полезно изучать, чтобы научиться настраивать функции MVC и наилучшим образом использовать имеющиеся варианты валидации. Адам подробно описывает библиотеку JQuery Validation в книге Pro jQuery (Apress, 2012).

# Выполнение удаленной валидации

Последняя функция валидации, которую мы рассмотрим в этой главе, - это удаленная валидация. Это техника валидации на стороне клиента, которая для выполнения валидации вызывает метод действия на сервере.

Распространенный пример удаленной валидации - это проверка доступности имени пользователя в приложениях, где такие имена должны быть уникальными; пользователь отправляет данные, и выполняется валидация на стороне клиента. Как часть этого процесса на сервер отправляется запрос Ajax, с помощью которого проводится валидация указанного имени пользователя. Если имя занято, то отображается ошибка валидации, чтобы пользователь мог ввести другое значение.

Это похоже на обычную валидацию на стороне сервера, но у этого подхода есть некоторые преимущества. Во-первых, удаленная валидация проводится только для отдельных свойств; преимущества валидации на стороне клиента все еще сохраняются для всех других значений данных. Во-вторых, запрос относительно легкий и ориентирован на валидацию, а не обработку всего объекта модели. Это означает, что мы можем сократить затраты производительности, которые требуются для этого запроса.

Третье отличие состоит в том, что удаленная валидация выполняется в фоновом режиме. Пользователь не должен нажимать на кнопку `Submit` и дожидаться визуализации нового представления. Это влияет на удобство работы с приложением, особенно в случае медленного соединения.

Таким образом, удаленная валидация является компромиссом; она позволяет нам найти баланс между клиентской валидацией и валидацией на стороне сервера. Хотя она не требует запросов к серверу, но и выполняется не так быстро, как обычная валидация на стороне клиента.

Чтобы использовать удаленную валидацию, для начала необходимо создать метод действия, который будет проверять одно из свойств модели. Мы будем проводить валидацию для свойства `Date` модели `Appointment`, чтобы гарантировать то, что указанная встреча состоится в будущем (это одно из первых правил валидации, которое мы использовали в начале главы, но его невозможно реализовать с помощью стандартных функций валидации на стороне клиента). В листинге 23-22 показан метод действия `ValidateDate`, который мы добавили в контроллер `Home`.

**Листинг 23-22:** Добавляем метод валидации в контроллер `Home`

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

namespace ModelValidation.Controllers
{
    public class HomeController : Controller
    {
```

```

public ViewResult MakeBooking()
{
    return View(new Appointment { Date = DateTime.Now });
}

[HttpPost]
public ViewResult MakeBooking(Appointment appt)
{
    if (ModelState.IsValid)
    {
        // statements to store new Appointment in a
        // repository would go here in a real project
        return View("Completed", appt);
    }
    else
    {
        return View();
    }
}

public JsonResult ValidateDate(string Date)
{
    DateTime parsedDate;
    if (!DateTime.TryParse(Date, out parsedDate))
    {
        return Json("Please enter a valid date (mm/dd/yyyy)",
                    JsonRequestBehavior.AllowGet);
    }
    else if (DateTime.Now > parsedDate)
    {
        return Json("Please enter a date in the future",
                    JsonRequestBehavior.AllowGet);
    }
    else
    {
        return Json(true, JsonRequestBehavior.AllowGet);
    }
}
}
}

```

Методы действий, которые поддерживают удаленную валидацию, должны возвращать тип JsonResult, который сообщает MVC Framework, что мы работаем с данными JSON. Кроме результата, метод валидации должен определить параметр с тем же именем, что и проверяемое поле данных. В этом примере это Date. Мы проверяем, возможно ли создать объект DateTime из значения, которое предоставил пользователь, и, если да, проверяем, что дата состоится в будущем.

#### *Подсказка*

*Мы могли бы использовать связывание данных и сделать параметром нашего метода действия объект DateTime, но в таком случае метод валидации не будет вызван, если пользователь введет что-то совершенно бессмыслиценное, например apple. Так произойдет потому, что механизм связывания не сможет создать объект DateTime из значения apple и сгенерирует исключение. Функция удаленной валидации не имеет возможности отобразить это исключение, и поэтому оно будет незаметно удалено. В результате поле ввода не будет подсвеченено, что создаст впечатление, что введенное пользователем значение является действительным. Лучше всего при работе с удаленной валидацией принять строковый параметр в методе действия и выполнить преобразование, анализ или связывание данных явно.*

Мы выражаем результаты валидации с помощью метода `Json`, который создает результат в формате JSON; этот результат будет проанализирован и обработан скриптом удаленной валидации на стороне клиента. Если обработанное значение отвечает нашим требованиям, то мы передаем `true` в метод `Json` в качестве параметра:

```
return Json(true, JsonRequestBehavior.AllowGet);
```

Если мы недовольны значением, то в качестве параметра передаем сообщение об ошибке валидации, которое должно быть показано пользователю, например:

```
return Json("Please enter a date in the future", JsonRequestBehavior.AllowGet);
```

В обоих случаях мы передаем в качестве параметра значение `JsonRequestBehavior.AllowGet`, потому что MVC Framework по умолчанию запрещает запросы GET, которые создают JSON, и это поведение необходимо переопределить. Без этого дополнительного параметра запросы валидации будут незаметно опускаться, и ошибки валидации не будут отображаться клиенту.

Чтобы использовать метод удаленной валидации в классе модели, мы применяем атрибут `Remote` к свойству, которое хотим проверить. В листинге 23-23 показано, как мы применили его к свойству `Date`.

#### Листинг 23-23: Используем атрибут `Remote` в классе модели

```
using System;
using System.ComponentModel.DataAnnotations;
using System.Web.Mvc;

namespace ModelValidation.Models
{
    public class Appointment
    {
        [Required]
        [StringLength(10, MinimumLength = 3)]
        public string ClientName { get; set; }

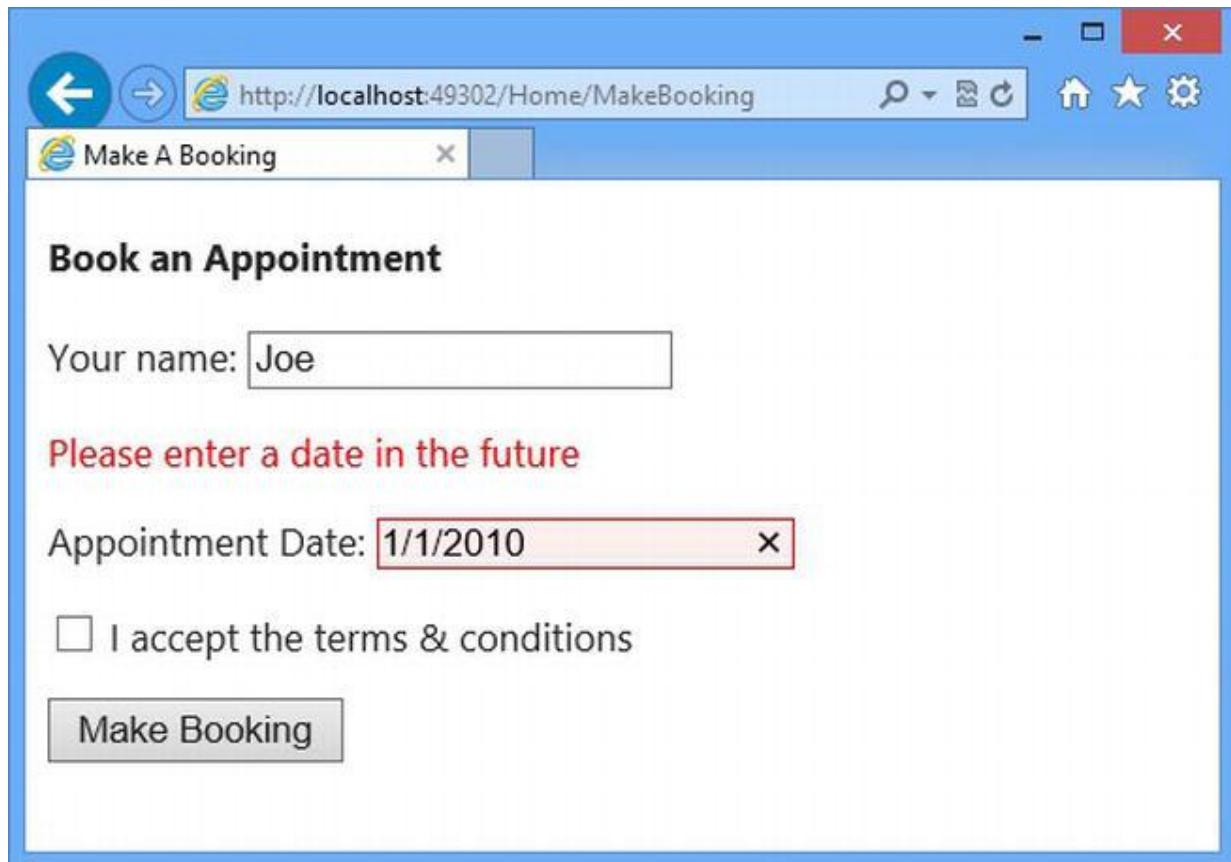
        [DataType(DataType.Date)]
        [Remote("ValidateDate", "Home")]
        public DateTime Date { get; set; }

        public bool TermsAccepted { get; set; }
    }
}
```

Аргументами для этого атрибута являются имя действия и контроллера для создания ссылки, с помощью которой библиотека валидации JavaScript будет выполнять валидацию; в нашем случае, это действие `ValidateDate` в контроллере `Home`.

Чтобы увидеть, как работает удаленная валидация, запустите приложение, перейдите по ссылке `/Home/MakeBooking` и введите дату, которая произошла в прошлом. Во время ввода появится сообщение валидации, как показано на рисунке 23-12.

Рисунок 23-12: Выполнение удаленной валидации



#### Внимание!

*Метод валидации будет вызван, когда пользователь отправит форму в первый раз, а затем каждый раз, когда он будет редактировать данные. В сущности, каждое нажатие клавиши будет отправлять вызов к серверу. В некоторых приложениях может получиться значительное число запросов, и это следует принимать во внимание, когда вы будете определять мощность и пропускную способность сервера, которая потребуется для данного приложения. Кроме того, вы можете не использовать удаленную валидацию для свойств, если для нее потребуется слишком много ресурсов (например, если вам придется отправлять запросы к медленному серверу, чтобы определить, является ли имя пользователя уникальным).*

## Резюме

В этой главе мы рассмотрели широкий спектр техник, с помощью которых можно выполнять валидацию модели и гарантировать, что предоставленные пользователем данные согласуются с ограничениями, определенными моделью данных. Валидация моделей – это очень важная тема, и правильная настройка валидации для приложений необходима, чтобы обеспечить пользователям удобство работы без сбоев. Не менее важным является то, что мы обеспечиваем целостность доменной модели и не сохраняем в системе нерелевантные или проблематичные данные.

# Связки (bundles) и режимы отображения

В этой главе мы рассмотрим две функции MVC Framework, которые упрощают разработку клиентской части приложения. Первая функция - связки - позволяет организовывать и оптимизировать файлы CSS и JavaScript, которые браузер запрашивает у сервера при обработке представлений и макетов. Вторая функция, режимы отображения, позволяет создавать представления, ориентированные на различные типы устройств.

## Понимание скриптовых библиотек по умолчанию

Некоторые из функций, которые мы рассмотрим в этой главе, относятся к управлению файлами JavaScript. Когда вы создаете проект MVC на любом шаблоне, кроме Empty, Visual Studio добавляет набор библиотек JavaScript в папку `Scripts`. Эти библиотеки, которые наиболее широко используются для разработки расширенной клиентской функциональности для приложений на стороне клиента, описаны в таблице 24-1.

**Таблица 24-1:** Библиотеки разработки на стороне клиента в папке `Scripts`

Имя файла	Описание
<code>jQuery-1.7.1.js</code>	Библиотека jQuery, которая облегчает работу с элементами HTML в браузере, особенно по сравнению со встроенными API, которые являются частью стандартов HTML.
<code>jquery-ui-1.8.20.js</code>	Библиотека jQuery UI, которая создает удобные пользовательские элементы управления из элементов HTML, позволяя создавать красивые пользовательские интерфейсы для веб-приложений. jQuery UI построена на jQuery.
<code>jquery.mobile-1.1.0.js</code>	Библиотека jQuery Mobile создает удобные пользовательские элементы управления для мобильных устройств. jQuery Mobile построена на jQuery и будет добавлена только в проекты, которые созданы на шаблоне Mobile.
<code>jquery-validate.js</code>	Библиотека jQuery Validation выполняет проверку ввода для элементов HTML <code>form</code> .
<code>knockout-2.1.0.js</code>	Knockout применяет шаблон Model-View-ViewModel к клиентской части приложения, который разделяет данные в клиентских приложениях и элементы, которые отображают их пользователю. Knockout часто называют «MVC для браузера».
<code>modernizr-2.5.3.js</code>	Modernizr обнаруживает в браузерах поддержку HTML5 и CSS3; это позволяет использовать новейшие функции, когда они доступны, и использовать прежнюю функциональность, если нет.

Здесь мы не собираемся рассматривать, как использовать эти библиотеки, потому что, с одной стороны, эта книга посвящена платформе на стороне сервера, с другой – библиотеки jQuery являются достаточно серьезной самостоятельной темой. Мы рекомендуем с ними ознакомиться, потому что они предоставляют более простые и надежные техники для разработки веб-приложений.

### Примечание

Признаем, что наше мнение предвзято. Стив создал библиотеку Knockout и Адам много писал об этой библиотеке (и о клиентской веб-разработке в целом) в своих книгах *Pro jQuery* (Apress, 2012) и *Pro JavaScript for Web App Development* (Apress, 2012). Для всех этих библиотек JavaScript

существуют альтернативы. Несмотря на то, что мы так любим те, которые Microsoft включила в новые проекты MVC, мы уверены, что если вам не понравится одна из библиотек в таблице, вы сможете найти ей замену, которая вас удовлетворит.

В дополнение к популярным библиотекам, которые показаны в таблице 24-1, папка `Scripts` содержит дополнительные библиотеки, которые поддерживают функции, специфические для Visual Studio или MVC. Они описаны в таблице 24-2.

**Таблица 24-2:** Специфические библиотеки для Visual Studio и MVC в папке `Scripts`

Имя файла	Описание
<code>jQuery-1.7.1.intellisense.js</code>	С ее помощью Visual Studio завершает имена функций во время записи кода jQuery в файлах представлений.
<code>jQuery.unobtrusive-ajax.js</code>	С ее помощью MVC Framework поддерживает функцию ненавязчивого Ajax, которая была описана в главе 21. Построена на jQuery.
<code>jQuery.validate-vsdoc.js</code>	С ее помощью Visual Studio завершает имена функций во время записи кода jQuery, который использует библиотеку jQuery Validation.
<code>jQuery.validate.unobtrusive.js</code>	С ее помощью MVC Framework поддерживает функцию ненавязчивой валидации, которую мы описали в главе 23. Построена на библиотеке jQuery Validation.

С файлами скриптов, которые поддерживают завершение кода в Visual Studio, не нужно делать ничего особенного. Когда они указаны в проекте приложения, Visual Studio находит и использует их автоматически. Ненавязчивый Ajax и библиотека валидации были описаны в главах 21 и 23. Эти скрипты как бы служат мостом между MVC Framework и функциональностью jQuery, на которой они построены.

В папке `Scripts` для многих из файлов, описанных в таблицах 24-1 и 24-2, есть две версии - обычная и минимизированная. Обычная версия содержит код JavaScript с комментариями, пробелами и значимыми именами функций и переменных. Они прекрасно подходят для отладки, так как вы можете прочитать исходный код и найти источник проблемы (одно из преимуществ JavaScript состоит в том, что его исходный код легко читается, потому что файлы JavaScript не компилируются до тех пор, пока не будут доставлены в браузер).

Проблема обычных файлов JavaScript в том, что они большие. Все полезные комментарии и значимые имена переменных и функций занимают место в файле, а следовательно, загрузка кода JavaScript займет больше времени и больше пропускной способности сервера.

Минимизированные файлы содержат ту же функциональность, что и соответствующие обычные файлы, но все читаемые имена, комментарии и пробелы были из них удалены, чтобы уменьшить размер файла. Чтобы вы лучше понимали эффект, отметим, что обычная библиотека jQuery, которую Visual Studio добавляет в новый проект, весит 252КБ, уменьшенная версия - 92КБ. Вы можете удивиться, почему кого-то-то будет волновать разница в 160Кб, но если вы обслуживаете миллионов запросов в день, разница станет очевидна – как в количестве времени, которое пользователь будет ждать загрузки кода JavaScript, так и в пропускной способности, которая потребуется для серверов.

### Подсказка

*Минимизированные файлы JavaScript по-прежнему содержат текст, но его очень трудно читать. Чтобы понять, что мы имеем в виду, просто откройте одну из минимизированных библиотек и просмотрите ее содержимое (это можно сделать из Visual Studio).*

Недостатком минимизированных файлов является то, что они затрудняют поиск проблем, поэтому во время разработки, как правило, используются обычные файлы, которые затем заменяются минимизированными версиями во время развертывания приложения.

Минимизированные версии файлов, как правило, содержат в имени фрагмент `.min`; например, минимизированная версия `jQuery-1.7.1.js` – это файл `jQuery-1.7.1.min.js`. Однако не все библиотеки следуют этому соглашению; например, файл `knockout-2.1.0.js` является минимизированным, а `knockout-2.1.0.debug.js` содержит читаемый код.

MVC Framework содержит несколько полезных функций, которые помогут вам управлять файлами JavaScript, которые использует ваше приложение. Теперь, когда мы познакомились с файлами скриптов и тем, что они делают, перейдем к созданию примера приложения, в котором затем и продемонстрируем эти функции управления.

## Подготовка приложения для примера

Для этой главы мы создали новый проект MVC под названием `ClientFeatures` на шаблоне `Basic`. Мы будем работать с приложением, аналогичным приложению из предыдущей главы, поэтому для начала создадим класс под названием `Appointment.cs` в папке `Models`. Содержимое этого файла показано в листинге 24-1.

#### Листинг 24-1: Класс модели `Appointment`

```
using System;
using System.ComponentModel.DataAnnotations;

namespace ClientFeatures.Models
{
    public class Appointment
    {
        [Required]
        public string ClientName { get; set; }

        [DataType(DataType.Date)]
        public DateTime Date { get; set; }

        public bool TermsAccepted { get; set; }
    }
}
```

Мы создали контроллер `Home`, который работает с классом модели `Appointment`; он показан в листинге 24-2.

#### Листинг 24-2: Контроллер `Home` в приложении `ClientFeatures`

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

```

namespace ClientFeatures.Controllers
{
    public class HomeController : Controller
    {
        public ViewResult MakeBooking()
        {
            return View(new Appointment
            {
                ClientName = "Adam",
                Date = DateTime.Now.AddDays(2),
                TermsAccepted = true
            });
        }

        [HttpPost]
        public JsonResult MakeBooking(Appointment appt)
        {
            // statements to store new Appointment in a
            // repository would go here in a real project
            return Json(appt, JsonRequestBehavior.AllowGet);
        }
    }
}

```

В этом контроллере есть две версии метода MakeBooking. Версия без параметров создает объект Appointment и передает его в метод View, который визуализирует представление по умолчанию.HttpPost версия метода MakeBooking ждет, когда механизм связывания создаст объект Appointment, кодирует его в методе Json и отправляет его обратно к клиенту в формате JSON.

В этой главе мы уделяем основное внимание функциям MVC Framework, которые поддерживают клиентскую разработку, так что мы немного сократили код контроллера, чего не стали бы делать в реальном проекте. Обратите внимание, что мы не выполняем никакой валидации, когда получаем запрос HTTP POST, и просто отправляем данные объекта, созданного механизмом связывания, в браузер в формате JSON (без поддержки ответов HTML на запросы POST).

Мы хотим максимально упростить использование формы с Ajax, которую мы определили в файле /Views/Home/MakeBooking.cshtml; он показан в листинге 24-3. Мы сосредоточимся на скриптах и элементах ссылок в представлении, так что взаимодействие с приложением отойдет на второй план.

### Листинг 24-3: Представление MakeBooking

```

@model ClientFeatures.Models.Appointment

@{
    ViewBag.Title = "Make A Booking";
    AjaxOptions ajaxOpts = new AjaxOptions
    {
        OnSuccess = "processResponse"
    };
}

<h4>Book an Appointment</h4>

<link rel="stylesheet" href="~/Content/CustomStyles.css" />
<script src="~/Scripts/jquery-1.7.1.min.js"></script>
<script src="~/Scripts/jquery.validate.js"></script>
<script src="~/Scripts/jquery.validate.unobtrusive.js"></script>
<script src="~/Scripts/jquery.unobtrusive-ajax.js"></script>

<script type="text/javascript">
    function processResponse(appt) {
        $('#successClientName').text(appt.ClientName);
    }
</script>

```

```

$( '#successDate' ).text( processDate( appt.Date ) );
switchViews();
}

function processDate(dateString) {
    return new Date( parseInt(dateString.substr(6,
        dateString.length - 8))).toDateString();
}

function switchViews() {
    var hidden = $('.hidden');
    var visible = $('.visible');
    hidden.removeClass("hidden").addClass("visible");
    visible.removeClass("visible").addClass("hidden");
}

$(document).ready(function () {
    $('#backButton').click(function (e) {
        switchViews();
    });
});

```

</script>

```

<div id="formDiv" class="visible">
@using (Ajax.BeginForm.ajaxOpts)
{
    @Html.ValidationSummary(true)
    <p>@Html.ValidationMessageFor(m => m.ClientName)</p>
    <p>Your name: @Html.EditorFor(m => m.ClientName)</p>
    <p>@Html.ValidationMessageFor(m => m.Date)</p>
    <p>Appointment Date: @Html.EditorFor(m => m.Date)</p>
    <p>@Html.ValidationMessageFor(m => m.TermsAccepted)</p>
    <p>@Html.EditorFor(m => m.TermsAccepted) I accept the terms & conditions</p>
    <input type="submit" value="Make Booking" />
}
</div>

<div id="successDiv" class="hidden">
<h4>Your appointment is confirmed</h4>
<p>Your name is: <b id="successClientName"></b></p>
<p>The date of your appointment is: <b id="successDate"></b></p>
<button id="backButton">Back</button>
</div>

```

В этом представлении есть два элемента `div`. Первый будет показан пользователю, когда представление будет визуализировано впервые; он содержит форму с Ajax. Когда форма отправлена, ответ сервера на запрос Ajax скроет форму и отобразит другой элемент `div`, в котором мы отображаем информацию о подтверждении записи на прием.

Мы включили в это представление ряд стандартных библиотек JavaScript из папки `Scripts` и определили локальный элемент `script`, который содержит простой код jQuery, специфичный для этого представления. Мы также добавили элемент `link`, который загружает из папки `/Content` файл CSS под названием `CustomStyles.css`; он показан в листинге 24-4.

#### Листинг 24-4: Содержимое файла `CustomStyles.css`

```

div.hidden { display: none; }
div.visible { display: block; }

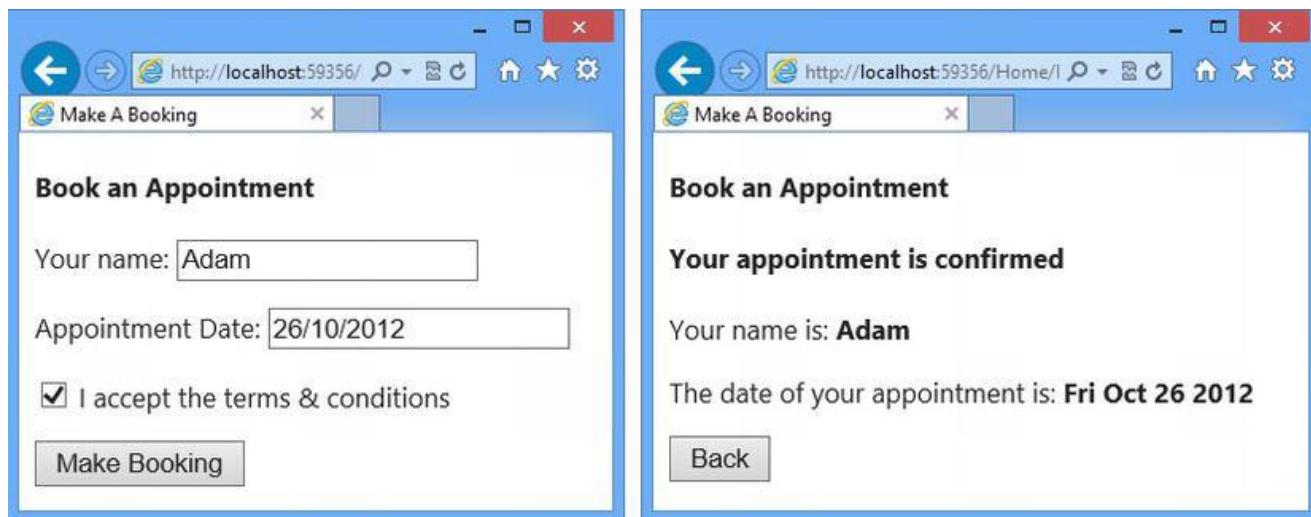
```

Мы хотим создать типичный сценарий для сложного представления, не усложняя при этом приложение. Именно поэтому мы добавили файл CSS, который содержит только два стиля, и используем столько библиотек jQuery для такого простого представления. Основная идея в том, что

нам придется управлять большим количеством файлов. Когда вы будете работать с реальными приложениями, вы будете поражены, как много файлов скриптов и стилей используют ваши представления.

Чтобы увидеть, как работает наш пример приложения, запустите его и перейдите по ссылке /Home/MakeBooking. Форма предварительно заполнена данными, так что вы можете просто нажать кнопку Make Booking, чтобы данные формы были отправлены на сервер с помощью Ajax. Когда будет получен ответ, вы увидите информацию об объекте Appointment, который был создан механизмом связывания из данных формы, а также кнопку, которая вернет вас обратно к форме, как показано на рисунке 24-1.

**Рисунок 24-1:** Используем пример приложения



## Управление скриптами и таблицами стилей

Представление, которое мы создали в листинге 24-3, является типичным для реальных проектов MVC. Оно содержит набор библиотек из папки Scripts, таблицы стилей CSS из папки Content, локальные элементы script и, конечно, разметку HTML и Razor.

Разработчики обычно пишут файлы представлений так же, как бы они писали страницы HTML; это нормальный подход, но не самый эффективный. Как мы покажем вам в следующих разделах, в представлении MakeBooking.cshtml есть скрытые проблемы. Можно сделать ряд улучшений относительно того, как мы управляем скриптами и таблицами стилей.

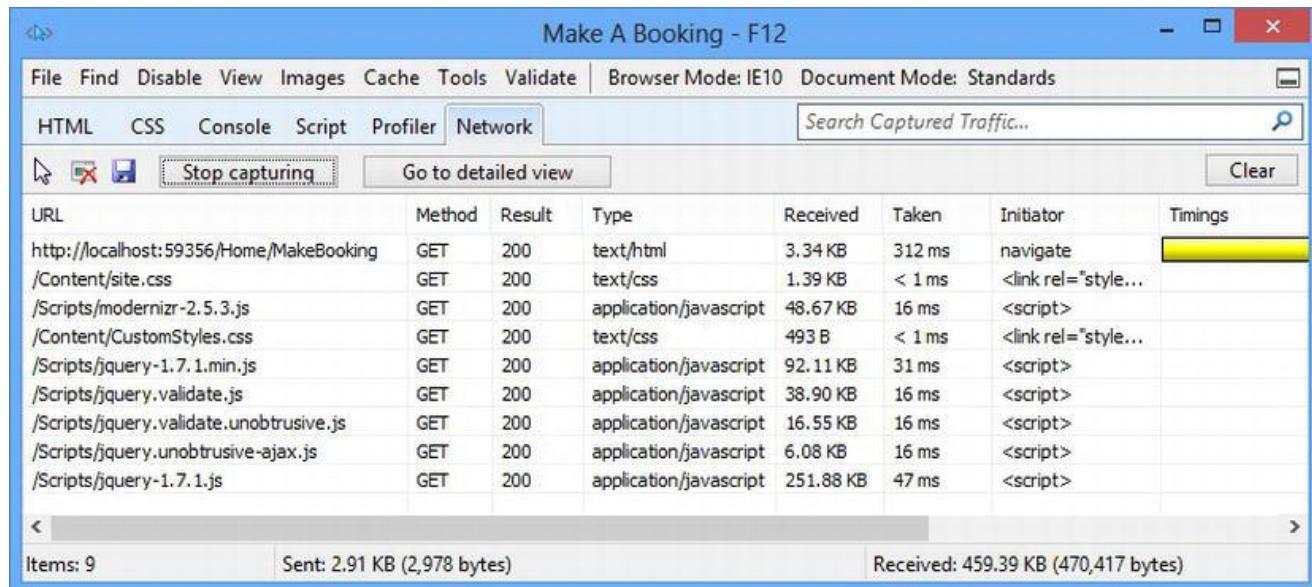
### Измеряем время загрузки скриптов и таблиц стилей

Если вы рассматриваете возможность оптимизации в любом проекте, начинать необходимо с измерений. Мы обеими руками за эффективность и оптимизацию приложений, но, как показывает опыт, из-за спешки и стремления оптимизировать совершенно незначительные проблемы, разработчики реализуют решения, которые вызывают проблемы в будущем.

Для проблем, которые мы рассмотрим в этой главе, мы будем выполнять измерения с помощью утилит F12 из Internet Explorer 10 (называемых так потому, что к ним можно получить доступ, нажав клавишу F12). Загрузите приложение, перейдите по ссылке /Home/MakeBooking и нажмите клавишу F12. Когда откроется окно утилит, перейдите на вкладку Network и нажмите кнопку Start

Capturing. Обновите содержимое вкладки браузера (щелкните правой кнопкой мыши в окне браузера и выберите Refresh), и вы увидите результаты, показанные на рисунке 24-2.

**Рисунок 24-2:** Измеряем время загрузки скриптов и таблиц стилей для примера приложения



Утилиты F12 в IE10 позволяют измерять производительность для сетевых запросов, которые отправляет ваше приложение. (Если вы не используете Internet Explorer, можно найти и другие инструменты. Наш любимый - Fiddler, который вы можете скачать бесплатно с [www.fiddler2.com](http://www.fiddler2.com).)

Чтобы сравнивать результаты оптимизации, которую мы проведем в этой главе, мы будем использовать данные, приведенные на рисунке 24-2, как отправную точку. Вот ключевые цифры:

- Браузер сделал 9 запросов по ссылке /Home/MakeBooking
- Было отправлено 2 запроса к файлам CSS.
- Было отправлено 6 запросов к файлам JavaScript.
- В общей сложности 2,978 байт было отправлены от браузера к серверу.
- В общей сложности 470,417 байт было отправлено от сервера к браузеру.

Это наихудшие данные о производительности для нашего приложения, так как перед тем, как перезагрузить представление, мы очистили кэш браузера. Мы сделали это, чтобы провести измерения для конкретной отправной точки, хотя мы и знаем, что в реальных условиях результаты были бы лучше, так как браузер кэширует файлы для предыдущих запросов.

Если мы перезагрузим страницу /Home/MakeBooking без очистки кэша, то мы получим следующие результаты:

- Браузер сделал 9 запросов по ссылке /Home/MakeBooking
- Было отправлено 2 запроса к файлам CSS.
- Было отправлено 6 запросов к файлам JavaScript.
- В общей сложности 2,722 байт было отправлены от браузера к серверу.
- В общей сложности 6,302 байт было отправлено от сервера к браузеру.

Это наш лучший сценарий, где все запросы к файлам CSS и JavaScript были обслужены с помощью ранее закэшированных файлов.

### **Примечание**

*В реальном проекте мы бы остановились и начали разбираться, есть ли здесь проблема, которую можно оптимизировать. Может показаться, что 470К – слишком много пропускной способности для простой веб-страницы, но все зависит от условий. Мы могли бы разрабатывать приложение для Инtranет, где канал дешевый и широкий, и любая оптимизация будет неоправданна из-за стоимости труда разработчика, который мог бы работать над более важным проектом. Равным образом, мы могли бы быть писать приложение, которое работает через интернет с важными для нас клиентами, у которых низкая скорость соединения; в этом случае стоит потратить время на оптимизацию каждого аспекта приложения. Важно то, что вы не должны автоматически оптимизировать в приложении все, что можно – чаще всего вы сможете найти себе более полезное занятие. (Особенно если вы украдкой проводите оптимизацию приложения, никому ничего не сказав. Скрытая оптимизация – это плохая идея, и конечном счете она только навредит вашей работе.)*

---

Если вы посмотрите на список файлов JavaScript, которые загружаются для представления, то заметите, что мы воссоздали две очень распространенных проблемы. Первая заключается в том, что мы смешали минимизированные и обычные файлы JavaScript. Это не очень большая проблема, но из-за нее отладка кода библиотеки в процессе разработки будет затруднена.

Второй проблемой является то, что мы загружаем и минимизированные, и обычные версии библиотек jQuery. Так происходит потому, что макет тоже загружает файлы JavaScript и CSS, и из-за отсутствия координации браузеру приходится загружать код, который у него уже есть.

Эти проблемы встречаются довольно часто. Далее мы рассмотрим функции MVC Framework, с помощью которых можно контролировать файлы скриптов и CSS. Мы также покажем вам, как сократить количество запросов, которое браузер должен отправить к серверу, и объем данных, который он должен загрузить.

## **Использование связок скриптов и стилей**

Первым делом мы объединим наши файлы JavaScript и CSS в связки, что позволит работать с ними как с одним элементом. Связки определяются в файле /App\_Start/BundleConfig.cs. В листинге 24- показано содержимое этого файла по умолчанию, которое создается Visual Studio.

**Листинг 24-5:** Содержимое файла BundleConfig.cs по умолчанию

```
using System.Web;
using System.Web.Optimization;

namespace ClientFeatures
{
    public class BundleConfig
    {
        public static void RegisterBundles(BundleCollection bundles)
        {
            bundles.Add(new ScriptBundle("~/bundles/jquery").Include(
                "~/Scripts/jquery-{version}.js"));
            bundles.Add(new ScriptBundle("~/bundles/jqueryui").Include(
                "~/Scripts/jquery-ui-{version}.js"));
            bundles.Add(new ScriptBundle("~/bundles/jqueryval").Include(
                "~/Scripts/jquery.unobtrusive-ajax.js"));
        }
    }
}
```

```

        "~/Scripts/jquery.unobtrusive*",
        "~/Scripts/jquery.validate*"));
bundles.Add(new ScriptBundle("~/bundles/modernizr").Include(
    "~/Scripts/modernizr-*"));
bundles.Add(new StyleBundle("~/Content/css").Include("~/Content/site.css"));
bundles.Add(new StyleBundle("~/Content/themes/base/css").Include(
    "~/Content/themes/base/jquery.ui.core.css",
    "~/Content/themes/base/jquery.ui.resizable.css",
    "~/Content/themes/base/jquery.ui.selectable.css",
    "~/Content/themes/base/jquery.ui.accordion.css",
    "~/Content/themes/base/jquery.ui.autocomplete.css",
    "~/Content/themes/base/jquery.ui.button.css",
    "~/Content/themes/base/jquery.ui.dialog.css",
    "~/Content/themes/base/jquery.ui.slider.css",
    "~/Content/themes/base/jquery.ui.tabs.css",
    "~/Content/themes/base/jquery.ui.datepicker.css",
    "~/Content/themes/base/jquery.ui.progressbar.css",
    "~/Content/themes/base/jquery.ui.theme.css"));
}
}
}

```

Статический метод RegisterBundles вызывается из метода Application\_Start файла Global.asax при первом запуске приложения MVC Framework. Метод RegisterBundles принимает объект BundleCollection, в котором мы регистрируем новые связки файлов с помощью метода Add.

#### **Подсказка**

*Классы, которые используются для создания связок, содержатся в пространстве имен System.Web.Optimization; во время написания этой книги документацию MSDN API для этого пространства имен было не так легко найти. Чтобы узнать больше о классах этого пространства имен, воспользуйтесь ссылкой <http://msdn.microsoft.com/en-us/library/system.web.optimization.aspx>.*

---

Мы можем создавать связки для скриптов и таблиц стилей, но имейте в виду, нельзя объединять эти типы файлов, так как MVC Framework оптимизирует их по-разному. Стили представлены в классе StyleBundle, скрипты – в классе ScriptBundle.

Чтобы создать новую связку, нужно создать экземпляр либо StyleBundle, либо ScriptBundle, оба из которых принимают единственный аргумент конструктора – это маршрут для ссылки на связку. Этот маршрут будет использоваться в качестве URL, по которому браузер будет запрашивать содержимое связки; поэтому важно использовать схему маршрута, чтобы он не конфликтовал с другими маршрутами в приложении. Самый безопасный способ это сделать – начать маршрут с ~/bundles или ~/Content. (Вы поймете, как это важно, когда мы объясним принцип работы связок).

Когда вы создали объект StyleBundle или ScriptBundle, добавьте в связку информацию о таблицах стилей и скриптах с помощью метода Include. Есть несколько интересных функций, которые позволяют сделать связки более гибкими. Чтобы продемонстрировать это, мы отредактировали и упростили связки нашего приложения в файле BundleConfig.cs, как показано в листинге 24-6. Мы добавили одну новую связку, отредактировали одну из существующих и удалили все те, которые нам не нужны.

#### **Листинг 24-6:** Настраиваем конфигурацию связок

```

using System.Web;
using System.Web.Optimization;

```

```

namespace ClientFeatures
{
    public class BundleConfig
    {
        public static void RegisterBundles(BundleCollection bundles)
        {
            bundles.Add(new StyleBundle("~/Content/css").Include("~/Content/*.css"));

bundles.Add(new ScriptBundle("~/bundles/clientfeaturesscripts")
    .Include("~/Scripts/jquery-{version}.js",
        "~/Scripts/jquery.validate.js",
        "~/Scripts/jquery.validate.unobtrusive.js",
        "~/Scripts/jquery.unobtrusive-ajax.js"));
        }
    }
}

```

Сначала мы изменили связку `StyleBundle` с маршрутом `~/Content/css`. Мы хотим, чтобы эта связка содержала все файлы CSS в нашем приложении, поэтому мы изменили аргумент, переданный в метод `Include`, с `~/Content/site.css` (который ссылается на один файл) на `~/Content/*.css`. Символ звездочки (\*) является универсальным, что означает, что теперь в нашей связке есть ссылка на все файлы CSS из папки `/Content` нашего проекта. Так можно гарантировать, что все файлы в папке будут автоматически включены в связку и порядок, в котором эти файлы будут загружены, не будет иметь значения. Так как для нас порядок загрузки CSS файлов не важен, мы можем использовать универсальный символ, но если вы будете полагаться на правила предшествования стилей CSS, то вам необходимо перечислить файлы по отдельности в определенном порядке.

Мы также добавили в файл `BundleConfig.cs` связку `ScriptBundle` с маршрутом `~/bundles/clientfeaturesscripts`. Вскоре вы опять увидите маршруты для этих связок, когда мы будем применять их в приложении. Для этой связки мы использовали метод `Include`, в котором указали отдельные файлы JavaScript, разделенные запятыми, потому что нам нужны только некоторые файлы из папки `Scripts` и для нас важен порядок, в котором браузер загружает и выполняет код.

Обратите внимание, как мы указали файл библиотеки jQuery:

`~/Scripts/jquery-{version}.js`

В имени файла очень удобно использовать фрагмент `{version}`, потому что в таком случае он будет соответствовать любой версии указанного файла и в зависимости от конфигурации приложения выберет либо обычную, либо минимизированную версию файла. MVC 4 содержит версию библиотеки jQuery 1.7.1, что означает, что наша связка будет включать файл `/Scripts/jquery-1.7.1.js` в процессе разработки и `/Scripts/jquery-1.7.1.min.js` при развертывании.

#### *Подсказка*

*Выбор между обычной и минимизированной версией осуществляется на основе узла compilation в файле `Web.config`. Обычная версия будет использоваться, если атрибут `debug` содержит значение `true`, а минимизированная - если атрибут `debug` содержит `false`. Далее в этой главе мы переключим режим приложения с отладки на развертывание, и вы сможете увидеть, как это происходит.*

Преимущество использования `{version}` заключается в том, что вы можете обновить библиотеки до последних версий и не определять связки заново. Недостатком является то, что `{version}` не сможет различить две версии одной и той же библиотеки в одном каталоге. Так, например, если мы добавим

файл `jQuery-1.7.2.js` в папку `Scripts`, то клиент загрузит файлы `1.7.1` и `1.7.2`. Чтобы это не подорвало нашу оптимизацию, мы должны убедиться, что в папке `/Scripts` находится только одна версия библиотеки.

#### Подсказка

*MVC Framework достаточно умен, чтобы игнорировать файлы IntelliSense при обработке `{version}` в связке, но мы всегда проверяем то, что запрашивает браузер, чтобы не включить нежелательные файлы. Вскоре мы покажем вам, как это сделать.*

## Применяем связки

Чтобы применить связки, сначала необходимо подготовить представление. Хотя этот шаг не является обязательным, он позволит MVC Framework выполнить максимальную оптимизацию нашего приложения. Мы создали новую папку `/Scripts/Home` и добавили в него файл JavaScript под названием `MakeBooking.js`. По соглашению, мы сохраняем файлы JavaScript для отдельных страниц в папках, названных по контроллеру. Содержимое файла `MakeBooking.js` показано в листинге 24-7.

#### Листинг 24-7: Содержимое файла `MakeBooking.js`

```
function processResponse(appt) {
    $('#successClientName').text(appt.ClientName);
    $('#successDate').text(processDate(appt.Date));
    switchViews();
}

function processDate(dateString) {
    return new Date(parseInt(dateString.substr(6,
        dateString.length - 8))).toDateString();
}

function switchViews() {
    var hidden = $('.hidden');
    var visible = $('.visible');
    hidden.removeClass("hidden").addClass("visible");
    visible.removeClass("visible").addClass("hidden");
}

$(document).ready(function () {
    $('#backButton').click(function (e) {
        switchViews();
    });
});
```

Это тот же самый код, который использовался ранее - мы только переместили его в отдельный файл. Следующим шагом будет изменение представления `/Views/Home/MakeBooking.cshtml`: мы удалим элементы `link` и `script`, для которых была создана связка, как показано в листинге 24-8. Мы хотим, чтобы браузер запрашивал только необходимые файлы, и если мы оставим эти элементы, запросы будут дублироваться. Единственный оставшийся элемент `script` ссылается на специфичный для данного представления файл `MakeBooking.js`, который мы создали в листинге 24-7.

#### Листинг 24-8: Удаляем элементы `link` и `script` из представления `MakeBooking.cshtml`

```
@model ClientFeatures.Models.Appointment

@{
    ViewBag.Title = "Make A Booking";
```

```

AjaxOptions ajaxOpts = new AjaxOptions
{
    OnSuccess = "processResponse"
};

<h4>Book an Appointment</h4>

<script src="~/Scripts/Home/MakeBooking.js" type="text/javascript"></script>

<div id="formDiv" class="visible">
    @using (Ajax.BeginForm(ajaxOpts))
    {
        @Html.ValidationSummary(true)
        <p>@Html.ValidationMessageFor(m => m.ClientName)</p>
        <p>Your name: @Html.EditorFor(m => m.ClientName)</p>
        <p>@Html.ValidationMessageFor(m => m.Date)</p>
        <p>Appointment Date: @Html.EditorFor(m => m.Date)</p>
        <p>@Html.ValidationMessageFor(m => m.TermsAccepted)</p>
        <p>@Html.EditorFor(m => m.TermsAccepted) I accept the terms & conditions</p>
        <input type="submit" value="Make Booking" />
    }
</div>

<div id="successDiv" class="hidden">
    <h4>Your appointment is confirmed</h4>
    <p>Your name is: <b id="successClientName"></b></p>
    <p>The date of your appointment is: <b id="successDate"></b></p>
    <button id="backButton">Back</button>
</div>

```

К связкам можно обращаться в файлах представлений, но мы, как правило, создаем связи только для контента, который является общим для нескольких представлений; это означает, что мы применяем связи в файлах макетов. В листинге 24-9 показан файл /Views/Shared/\_Layout.cshtml, который добавила в проект Visual Studio.

#### Листинг 24-9: Макет, добавленный в проект Visual Studio

```

<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>

    @Styles.Render("~/Content/css")
    @Scripts.Render("~/bundles/modernizr")
</head>
<body>
    @RenderBody()

    @Scripts.Render("~/bundles/jquery")

    @RenderSection("scripts", required: false)
</body>
</html>

```

Связки добавляются с помощью вспомогательных методов `@Scripts.Render` и `@Styles.Render`. Как видите, макет уже содержит три связи, которые выделены жирным шрифтом. Два вызова `@Scripts.Render` объясняют некоторые исходные данные профиля. Браузер запрашивает две копии библиотеки jQuery из-за связи `~/bundles/jquery`. Из-за связи `~/bundles/modernizr` браузер запрашивает библиотеку, которую мы не используем в приложении.

В самом деле, используется только связка `~/Content/css`, потому что она загружает файл `/Content/Site.css`, который был изначально определен в связке, и все файлы CSS в папке `/Content` в соответствии с изменениями, которые мы сделали в листинге 24-9. Чтобы создать необходимое нам поведение, мы отредактировали файл `_Layout.cshtml`, чтобы использовать наши новые связки и удалить ненужные ссылки, как показано в листинге 24-10.

#### Листинг 24-10: Добавляем нужные связки в файл `_Layout.cshtml`

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>

    @Styles.Render("~/Content/css")
</head>
<body>
    @RenderBody()

    @Scripts.Render("~/bundles/clientfeaturesscripts")

    @RenderSection("scripts", required: false)
</body>
</html>
```

Чтобы увидеть HTML, который генерируют эти вспомогательные методы, запустите приложение, перейдите по ссылке `/Home/MakeBooking` и просмотрите исходный код страницы. Вот вывод метода `Styles.Render` для связки `~/Content/css`:

```
<link href="/Content/CustomStyles.css" rel="stylesheet"/>
<link href="/Content/Site.css" rel="stylesheet"/>
```

А вот вывод метода `Scripts.Render`:

```
<script src="/Scripts/jquery-1.7.1.js"></script>
<script src="/Scripts/jquery.unobtrusive-ajax.js"></script>
<script src="/Scripts/jquery.validate.js"></script>
<script src="/Scripts/jquery.validate.unobtrusive.js"></script>
```

#### Используем секции scripts

Осталось еще одно изменение. Выполнение нашего специфического для представления JavaScript, который содержится в файле `/Scripts/Home/MakeBooking.js`, зависит от jQuery, которая должна создать обработчик события для кнопки. Следовательно, мы должны гарантировать, что файл jQuery загружается перед `MakeBooking.js`. Если вы посмотрите на макет в листинге 24-10, то увидите, что вызов метода `RenderBody` находится перед вызовом метода `Scripts.Render`, а это значит, что элемент `script` в представлении появляется перед элементами `script` в макете, и код кнопки не будет работать. (Он либо не создаст оповещения, либо сообщит об ошибке JavaScript, в зависимости от используемого браузера).

Это можно исправить, переместив вызов `Scripts.Render` в элемент `head` в представлении; по сути, так мы обычно и делаем. Тем не менее, мы можем также использовать дополнительную секцию `scripts`, которая определена в файле `_Layout.cshtml` и которую вы можете увидеть в листинге 24-10. В листинге 24-11 показано, как мы обновили представление `MakeBooking.cshtml`, чтобы использовать эту секцию.

#### Листинг 24-11: Используем дополнительную секцию scripts в представлении

```
@model ClientFeatures.Models.Appointment

@{
    ViewBag.Title = "Make A Booking";
    AjaxOptions ajaxOpts = new AjaxOptions
    {
        OnSuccess = "processResponse"
    };
}

<h4>Book an Appointment</h4>

@section scripts {
    <script src="~/Scripts/Home/MakeBooking.js" type="text/javascript"></script>
}

<div id="formDiv" class="visible">
    @using (Ajax.BeginForm(ajaxOpts))
    {
        @Html.ValidationSummary(true)
        <p>@Html.ValidationMessageFor(m => m.ClientName)</p>
        <p>Your name: @Html.EditorFor(m => m.ClientName)</p>
        <p>@Html.ValidationMessageFor(m => m.Date)</p>
        <p>Appointment Date: @Html.EditorFor(m => m.Date)</p>
        <p>@Html.ValidationMessageFor(m => m.TermsAccepted)</p>
        <p>@Html.EditorFor(m => m.TermsAccepted) I accept the terms & conditions</p>
        <input type="submit" value="Make Booking" />
    }
</div>

<div id="successDiv" class="hidden">
    <h4>Your appointment is confirmed</h4>
    <p>Your name is: <b id="successClientName"></b></p>
    <p>The date of your appointment is: <b id="successDate"></b></p>
    <button id="backButton">Back</button>
</div>
```

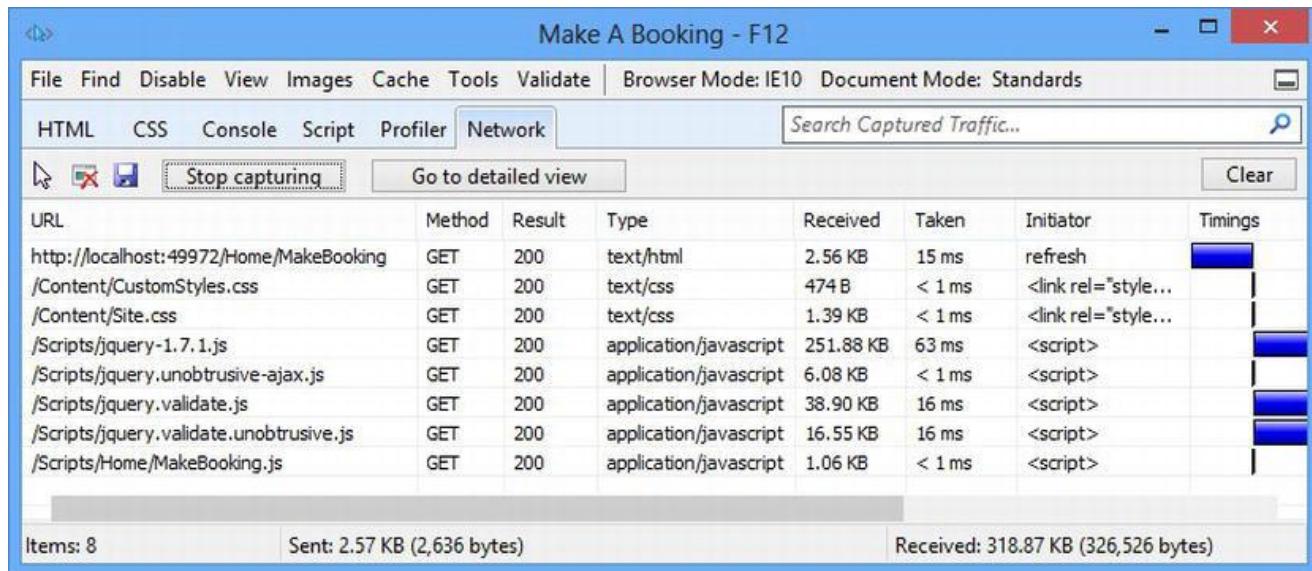
Секция scripts появляется после вызова Scripts.Render в макете, что означает, что наш специфический для представления скрипт не будет загружен до jQuery, и наш элемент button будет работать так, как мы планировали. При использовании связок эта ошибка возникает очень часто, и именно поэтому мы явно ее продемонстрировали.

## Измеряем эффект изменений

Мы определили собственные связки, удалили нежелательные ссылки на JavaScript и в целом упорядочили используемые скрипты и таблицы стилей. Сейчас настало время, чтобы измерить эффект изменений и оценить разницу.

Для этого мы очистили кэш браузера, перешли по ссылке /Home/MakeBooking и отследили запросы, которые были отправлены браузером, с помощью утилиты F12. Результат показан на рисунке 24-3.

**Рисунок 24-3:** Измеряем производительность приложения, в котором используются связки



Вот краткая информация о производительности:

- Браузер сделал 8 запросов по ссылке /Home/MakeBooking.
- Было отправлено 2 запроса к файлам CSS.
- Было отправлено 5 запросов к файлам JavaScript.
- В общей сложности 2,638 байт было отправлено от браузера к серверу.
- В общей сложности 326,549 байт было отправлено от сервера к браузеру.

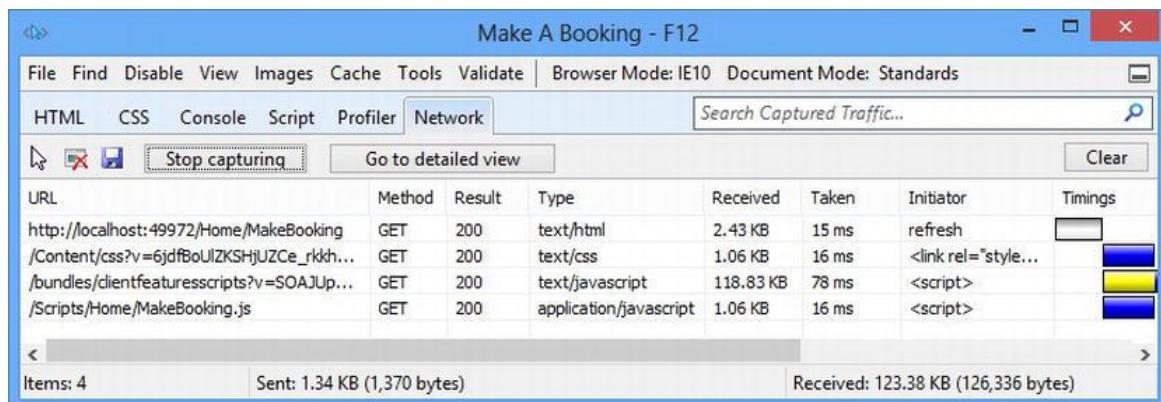
Это не плохо. Мы уменьшили объем данных, загружаемый браузером, примерно на 30 процентов. Но результаты станут еще лучше, если мы переключим приложение из режима отладки на развертывание. Для этого необходимо установить атрибуту `debug` в элементе `compilation` файла `Web.config` значение `false`, как показано в листинге 24-12.

**Листинг 24-12:** Отключаем режим отладки в файле `Web.config`

```
<system.web>
  <httpRuntime targetFramework="4.5" />
  <compilation debug="true" targetFramework="4.5" />
```

Когда это изменение внесено, мы перезапустим приложение, очистим кэш браузера и измерим сетевые запросы снова. Результат показан на рисунке 24-4.

**Рисунок 24-4:** Измеряем производительность приложения, в котором используются связки, в режиме развертывания



Вот краткая информация о производительности:

- Браузер сделал 4 запроса по ссылке /Home/MakeBooking.
- Был отправлен только 1 запрос к CSS.
- Было отправлено 2 запроса к файлам JavaScript.
- В общей сложности 1,372 байт было отправлено от браузера к серверу.
- В общей сложности 126,340 байт было отправлено от сервера к браузеру.

Вам может стать интересно, почему уменьшилось число запросов к файлам CSS и JavaScript.

Причина в том, что MVC Framework объединяет и минимизирует таблицы стилей и файлы JavaScript в режиме развертывания, так что все содержимое связки может быть загружено с помощью одного запроса. Чтобы увидеть, как это работает, посмотрите на HTML, который создает приложение. Вот код, сгенерированный методом `Styles.Render`:

```
<link href="/Content/css?v=6jdfBoU1ZKSHjUZCe_rkhh4S8jotNCGFD09Dym7kBWE1" rel="stylesheet"/>
```

А вот код, сгенерированный методом `Scripts.Render`:

```
<script src="/bundles/clientfeaturesscripts?v=SOAJUpvGwNr0XsILBsLrEwEpdyTziIN9frqxTjgTyWE1"></script>
```

Эти длинные URL используются, чтобы запросить содержимое связки в одном блоке данных. MVC Framework минимизирует данные CSS не так, как файлы JavaScript, поэтому мы должны объединять таблицы стилей и скрипты в разных связках.

Проведенная оптимизация производит значительный эффект. Браузер отправляет гораздо меньше запросов, что уменьшает объем данных, передаваемых клиенту. И мы передаем меньше данных обратно на сервер; в самом деле, мы отправили около 27% от того объема данных, который был зафиксирован при первом измерении производительности в начале этой главы.

На этом мы завершим оптимизацию запросов. Можно было бы добавить файл `MakeBooking.js` в связку, устранив еще один запрос и уменьшив количество кода; но мы достигли точки, где отдача начинает уменьшаться, и мы начинаем смешивать контент, специфический для представления, с кодом макета, чего лучше избегать. Если бы у нас было более сложное приложение, можно было бы создать вторую связку скриптов, в которую поместить больше пользовательского кода, но наше приложение очень простое и здесь решающим правилом оптимизации будет знать, где остановиться. В этом приложении мы достигли этой точки.

## Работа с мобильными устройствами

Вы будете часто оптимизировать приложения MVC Framework для мобильных устройств, а для этого необходимо адаптировать контент, который вы предоставляете пользователю, а также учитывать доступные функции и ограничения устройства. Между мобильными устройствами и настольными компьютерами есть много различий, но больше всего хлопот вызывают сенсорный ввод и ограниченный размер экрана. Чтобы в представлениях, которые визуализируются для мобильного клиента, можно было использовать сенсорный ввод, необходимы элементы большего размера и свободное пространство между ними, чтобы ввод был достаточно точным. Также необходимо убедиться, что контент можно прочитать на маленьком экране.

Мы не собираемся вдаваться в подробности разработки для мобильных устройств, но продемонстрируем, как MVC Framework помогает обеспечить поддержку обычных и мобильных клиентов в приложении.

#### **Подсказка**

*Когда вы создаете новый проект MVC Framework, доступна опция Mobile template, но это всего лишь вариант шаблона Internet, в котором добавлена поддержка библиотеки jQuery Mobile. Нам очень нравится jQuery Mobile, и, как вы уже догадались, Адам подробно описал ее в других своих книгах. Однако нам не нравится шаблон Mobile, поскольку он добавляет стандартные представления и контроллеры для аутентификации, которые, по нашему мнению, не очень хорошо работают. Вместо этого мы рекомендуем вам установить jQuery Mobile и другие необходимые библиотеки JavaScript с помощью NuGet.*

---

## **Пересматриваем приложение**

Лучше всего начать оптимизировать приложение для мобильных устройств с самого начала разработки проекта. Здесь мы будем делать все наоборот, а именно - добавим поддержку в существующее приложение ClientFeatures, на примере которого мы продемонстрировали, как использовать связи.

Для начала попытаемся понять, как наше приложение выглядит на различных мобильных устройствах. Для этого мы будем использовать эмулятор Opera Mobile Emulator, который можно бесплатно скачать с [www.opera.com/developer/tools/mobile](http://www.opera.com/developer/tools/mobile). Конечно, ничем нельзя заменить тестирование мобильного приложения на реальных мобильных устройствах, но Opera Mobile Emulator позволит быстро оценить, где мы находимся. Он распространяется бесплатно, обеспечивает верную эмуляцию браузера Opera Mobile, который используется на многих мобильных устройствах, и позволяет протестировать различные профили и возможности аппаратного обеспечения.

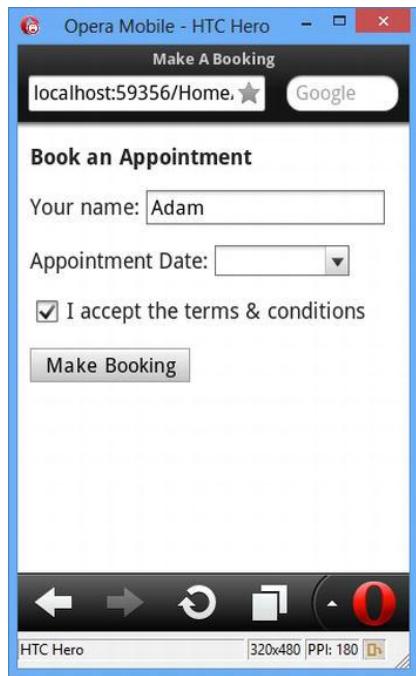
#### **Подсказка**

*Есть другие эмуляторы, в том числе для платформ Windows Phone, Android и Blackberry. Как правило, они работают довольно медленно и трудны в использовании, потому что они эмулируют целую мобильную операционную систему, а не только браузер. Мы еще не нашли ни одного вменяемого эмулятора iPhone для Windows PC: в них обычно используется Windows-версия Safari, которая отличается от версии iOS и похожа на неудачную копию браузера Apple.*

---

На рисунке 24-5 показано, как отображается в мобильном браузере представление /Home/MakeBooking. В эмуляторе мы использовали профиль HTC Hero, который имитирует типичное, ничем не выдающееся мобильное устройство с сенсорным дисплеем 320 × 480 пикселей и разрешением 180 пикселей на дюйм.

**Рисунок 24-5:** Приложение в мобильном браузере



Так как это очень простое приложение, оно не так уж плохо отображается на экране. Если придиаться, то можно сказать, что элементы расположены не лучшим образом для использования на сенсорном экране. Большинство реальных приложений, разработанных и протестированных в браузерах для РС, в мобильных браузерах выглядят ужасно.

## Используем специальные макеты и представления для мобильных устройств

В следующих разделах мы рассмотрим, как создавать мобильные версии представлений и макетов, в которых можно адаптировать контент для целого ряда устройств. Это совсем несложный процесс, и для обеспечения базовой поддержки достаточно создать версии представлений и макетов с фрагментом .Mobile в имени. Эта техника использует встроенную поддержку режимов отображения.

В качестве примера мы создали новый файл представления под названием Views/Home/MakeBooking.Mobile.cshtml. Обратите внимание на фрагмент .Mobile перед расширением файла. Содержимое представления показано в листинге 24-13.

### Листинг 24-13: Представление /Views/Home/MakeBooking.Mobile.cshtml

```
@model ClientFeatures.Models.Appointment

@{
    ViewBag.Title = "Make A Booking";
    AjaxOptions ajaxOpts = new AjaxOptions
    {
        OnSuccess = "processResponse"
    };
}

<h4>This is the MOBILE View</h4>

@section scripts {
    <script src="~/Scripts/Home/MakeBooking.js" type="text/javascript"></script>
}
```

```

<div id="formDiv" class="visible">
    @using (Ajax.BeginForm.ajaxOpts))
    {
        @Html.ValidationSummary(true)
        <p>@Html.ValidationMessageFor(m => m.ClientName)</p>
        <p>Name:</p><p>@Html.EditorFor(m => m.ClientName)</p>
        <p>@Html.ValidationMessageFor(m => m.Date)</p>
        <p>Date:</p><p>@Html.EditorFor(m => m.Date)</p>
        <p>@Html.ValidationMessageFor(m => m.TermsAccepted)</p>
        <p>@Html.EditorFor(m => m.TermsAccepted) Terms & Conditions</p>
        <input type="submit" value="Make Booking" />
    }
</div>

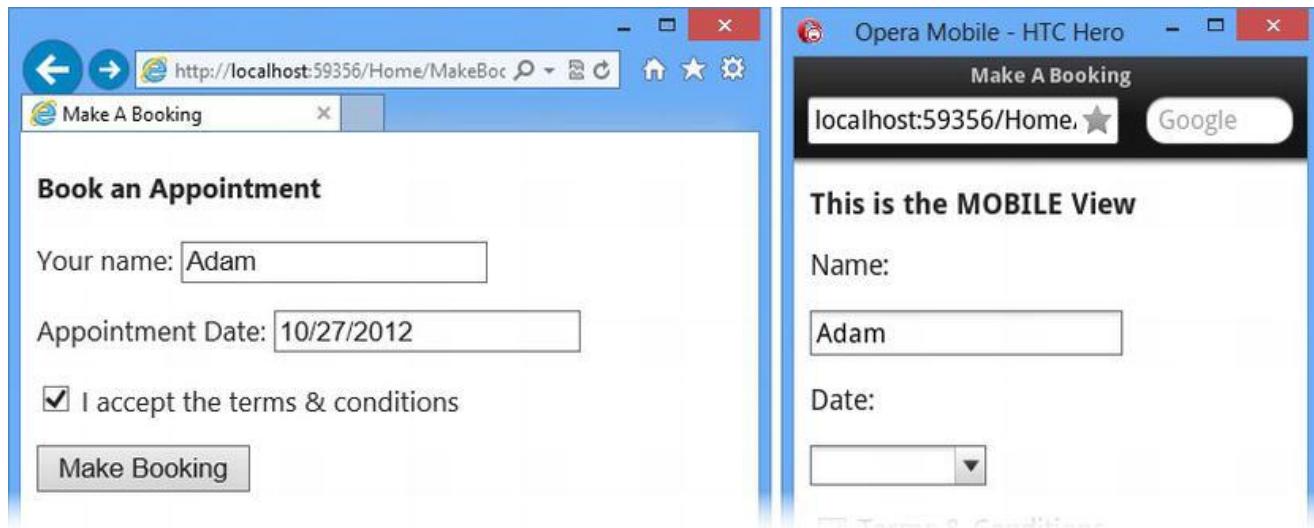
<div id="successDiv" class="hidden">
    <h4>Your appointment is confirmed</h4>
    <p>Your name is: <b id="successClientName"></b></p>
    <p>The date of your appointment is: <b id="successDate"></b></p>
    <button id="backButton">Back</button>
</div>

```

Мы внесли небольшие корректизы в это представление, чтобы метки для элементов input отображались отдельно, а также изменили содержание элемента h4 title, чтобы стало очевидно, какое представление отображается.

MVC Framework применяет мобильное представление автоматически. Если вы запустите приложение и перейдете по ссылке /Home/MakeBooking в браузере настольного компьютера, то увидите HTML, сгенерированный для представления /Views/Home/MakeBooking.cshtml. Но если вы перейдете по той же ссылке в эмуляторе мобильного браузера, то увидите HTML, сгенерированный для представления /Views/Home/MakeBooking.Mobile.cshtml, как показано на рисунке 24-6.

**Рисунок 24-6:** Переход по ссылке в настольном и мобильном браузерах



Способ обнаружения мобильных браузеров немного странный и опирается на набор текстовых файлов, которые входят в .NET Framework. Мы нашли их в C:\Windows\Microsoft.NET\Framework\v4.0.30319\Config\Browsers, но в других операционных системах директория может отличаться. Здесь находится ряд файлов, каждый из которых идентифицирует мобильный браузер – есть файл и для Opera Mobile. Когда MVC Framework получает запрос от мобильного эмулятора, он смотрит на строку userAgent, которую отправляют все браузеры, и определяет, что запрос был отправлен с мобильного устройства; далее используется представление MakeBooking.Mobile.cshtml.

## Создаем пользовательские режимы отображения

По умолчанию MVC Framework обнаруживает только мобильные устройства и рассматривает их в одной категории. Если вы хотите уточнить, как отвечать различным типам устройств, то можете создать свои собственные режимы отображения. Чтобы это продемонстрировать, мы будем использовать профиль Amazon Kindle Fire, который имеется в Opera Mobile Emulator. Opera Mobile, установленная на планшете, посыпает строку `UserAgent`, которая не соответствует тому, что ожидает .NET Framework от Opera Mobile; таким образом, к устройству отправляется стандартное представление `MakeBooking.cshtml`.

В листинге 24-14 показано, как мы изменили файл `Global.asax`, чтобы создать новый режим отображения для браузера Opera Tablet.

**Листинг 24-14:** Создаем новый режим отображения в файле `Global.asax`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Http;
using System.Web.Mvc;
using System.Web.Optimization;
using System.Web.Routing;
using System.Web.WebPages;

namespace ClientFeatures
{
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            DisplayModeProvider.Instance.Modes.Insert(0,
                new DefaultDisplayMode("OperaTablet")
            {
                ContextCondition = (context => context.Request.UserAgent.IndexOf
                    ("Opera Tablet", StringComparison.OrdinalIgnoreCase) >= 0)
            });
            AreaRegistration.RegisterAllAreas();
            WebApiConfig.Register(GlobalConfiguration.Configuration);
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);
        }
    }
}
```

Статическое свойство `DisplayModeProvider.Instance.Modes` возвращает коллекцию, с помощью которой мы можем определить пользовательские режимы отображения; для этого необходимо создать экземпляр объекта `DefaultDisplayMode` и установить в качестве значения свойства `ContextCondition` лямбда-выражение, которое получает объект `HttpContextBase` и возвращает логическое значение. Аргументом конструктора для класса `DefaultDisplayMode` является имя режима отображения, которое будет использоваться для поиска макетов и представлений, адаптированных для конкретного типа устройств. Мы указали `OperaTablet`, что означает, что MVC Framework будет искать такие представления, как `/Views/Home/MakeBooking.OperaTablet.cshtml`.

## Подсказка

Обратите внимание, что мы использовали метод `Insert`, чтобы поместить объект `DefaultDisplayMode` с нулевым индексом в коллекции, которую возвращает свойство `DisplayModeProvider.Instance.Modes`. MVC Framework проверяет режимы отображения по очереди и останавливает поиск, если выражение `ContextCondition` какого-либо режима возвращает `true`. Есть резервный режим отображения, который будет соответствовать любому запросу и использовать представление по умолчанию (т.е. без дополнений `OperaTable` или `Mobile`). Мы должны гарантировать, что наш режим отображения находится в очереди перед резервным вариантом.

Мы используем объект `HttpContextBase`, чтобы решить, соответствует ли полученный запрос искомому режиму отображения. Мы проверяем, содержит ли свойство `Request.UserAgent` значение `Opera Tablet`, и если да, то возвращаем `true`.

Чтобы воспользоваться этим режимом, мы создали новое представление под названием `/Views/Home/MakeBooking.OperaTablet.cshtml`, которое показано в листинге 24-15.

**Листинг 24-15:** Содержимое файла `MakeBooking.OperaTablet.cshtml`

```
@model ClientFeatures.Models.Appointment

{@
    ViewBag.Title = "Make A Booking";
    AjaxOptions ajaxOpts = new AjaxOptions
    {
        OnSuccess = "processResponse"
    };
}

<h4>This is the OPERA TABLET View</h4>

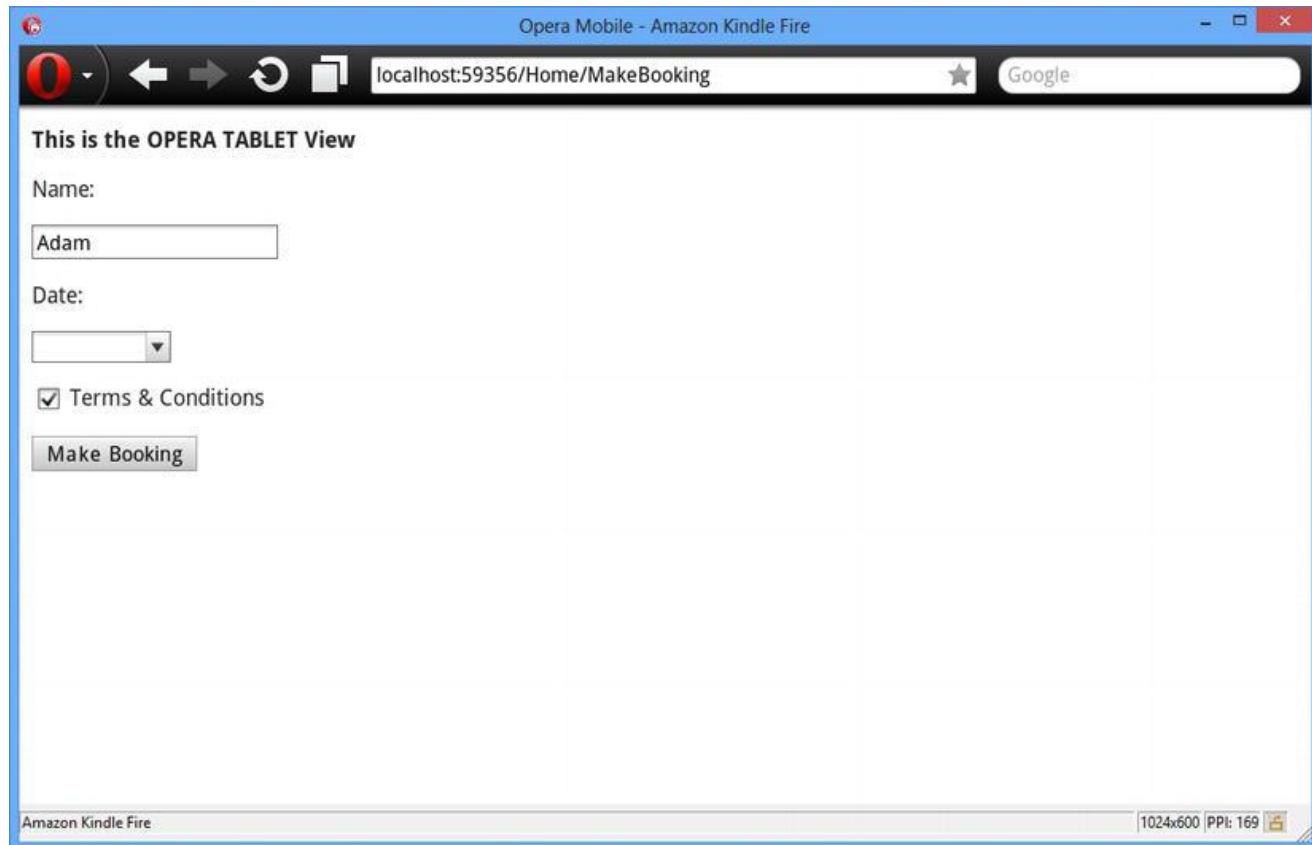
@section scripts {
    <script src="~/Scripts/Home/MakeBooking.js" type="text/javascript"></script>
}

<div id="formDiv" class="visible">
    @using (Ajax.BeginForm(ajaxOpts))
    {
        @Html.ValidationSummary(true)
        <p>@Html.ValidationMessageFor(m => m.ClientName)</p>
        <p>Name:</p><p>@Html.EditorFor(m => m.ClientName)</p>
        <p>@Html.ValidationMessageFor(m => m.Date)</p>
        <p>Date:</p><p>@Html.EditorFor(m => m.Date)</p>
        <p>@Html.ValidationMessageFor(m => m.TermsAccepted)</p>
        <p>@Html.EditorFor(m => m.TermsAccepted) Terms & Conditions</p>
        <input type="submit" value="Make Booking" />
    }
</div>

<div id="successDiv" class="hidden">
    <h4>Your appointment is confirmed</h4>
    <p>Your name is: <b id="successClientName"></b></p>
    <p>The date of your appointment is: <b id="successDate"></b></p>
    <button id="backButton">Back</button>
</div>
```

Здесь мы внесли только одно изменение - в элемент `h4`, который сообщает нам, какое используется представление. Если мы запустим Opera Mobile Emulator с профилем Kindle File и перейдем по ссылке `/Home/MakeBooking`, то увидим, что используется наше новое представление, как показано на рисунке 24-7.

**Рисунок 24-7:** Результат создания пользовательского режима отображения



Очевидно, что в реальном проекте мы сделали бы немного больше изменений, учитывая размер экрана и необходимость сенсорного ввода; но в этой главе мы остановимся на краткой демонстрации того, как использовать пользовательские режимы отображения для точного контроля над представлениями.

#### *Внимание!*

*Будьте осторожны при создании пользовательских режимов отображения. Очень легко захватить неправильные запросы от клиентов или пропустить тонкие различия между версиями браузера, работающими на конкретных типах мобильных устройств. Мы рекомендуем тщательное тестирование на различных устройствах.*

## Резюме

В этой главе мы рассмотрели связки и режимы отображения, которые могут быть полезны при разработке клиентской части веб-приложения. В следующей главе мы познакомим вас с Web API, с помощью которого можно легко создавать веб-сервисы для клиентов.

# Web API

В этой главе мы опишем Web API – новую функцию платформы ASP.NET, позволяющую легко и быстро создавать веб-службы, которые предоставляют API для HTTP-клиентов (известные как Web API).

Функция Web API имеет ту же основу, что и обычные приложения MVC Framework, но не является частью MVC. Microsoft дублировала некоторые ключевые классы и характеристики, связанные с пространством имен `System.Web.Mvc`, в пространство имен `System.Web.Http`. Таким образом, Web API является частью ядра платформы ASP.NET и может использоваться в других типах веб-приложений или работать как автономный движок веб-служб.

К счастью, нам уже хорошо знакомы основы Web API, потому что мы описали их в предыдущих главах. В этой главе мы покажем вам, как добавить Web API в обычный проект приложения MVC Framework. Этот процесс настолько прост, что большую часть времени мы затратим на создание кода JavaScript, с помощью которого будем использовать созданный API.

Невозможно недооценить то, как Web API упрощает создание веб-служб (это большой шаг вперед по сравнению с другими технологиями для работы с веб-службами Microsoft, которые появлялись за последнее десятилетие). Нам нравится Web API и мы рекомендуем вам использовать его в своих проектах, главным образом потому, что он очень прост и имеет ту же структуру, что и MVC Framework.

## Понимание Web API

Чтобы использовать функцию Web API, необходимо добавить специальный контроллер в приложение MVC Framework. Этот контроллер - *API Controller* - имеет две отличительные характеристики:

1. Методы действий возвращают модель, а не объекты `ActionResult`.
2. Методы действий выбираются в зависимости от метода HTTP, который использовался в запросе.

Объекты моделей, созданные методами действий контроллера API, преобразовываются в формат JSON и отправляются клиенту. Контроллеры API предназначены для доставки веб-данных сервисов, так что они не поддерживают представления, макеты или любые другие функции, которые создают HTML для отображения в браузере. Контроллер API может поддерживать запросы от любого клиента, но чаще всего он используется для обслуживания запросов Ajax от веб-приложений; это мы и продемонстрируем в данной главе.

Как мы показали в главе 21, методы действий, которые возвращают данные JSON для поддержки Ajax, можно создавать и в обычных контроллерах, но контроллер API предлагает альтернативный подход. Он позволяет отделить действия, которые возвращают данные, от действий, которые возвращают представления, и, таким образом, облегчает и ускоряет создание Web API общего назначения.

### *Подсказка*

*Вас никто не обязывает использовать контроллеры API (как и практически все остальные функции MVC Framework). В реальных проектах мы используем техники из главы 21 наряду с контроллерами API. Как правило, мы используем контроллеры API,*

---

*когда необходимо создать много действий, которые возвращают JSON, или когда мы работаем над проектом, который не имеет компонента HTML и предоставляет только сервисы данных.*

---

Чтобы объяснить, как работают контроллеры API, проще всего будет создать приложение MVC Framework, в котором будут использоваться такие контроллеры; это мы и сделаем позже в этой главе. Тем не менее, Web API - довольно простая функция, и опирается на многие возможности MVC, которые мы подробно рассмотрели в предыдущих главах, так что мы потратим большую часть времени на создание приложения, а в конце просто добавим к нему контроллер API.

С одной стороны, это даже хорошо. Это означает, что вы уже знаете все функции, на которые опирается контроллер API. Но пример все равно получается странным, потому что нам придется создать много стандартных компонентов приложения MVC, прежде чем мы перейдем к контроллеру API. К тому же, нам придется написать код JavaScript, который отправляет запросыAjax. Для этого мы будем использовать jQuery, но не будем разбирать код подробно, а только объясним, что делает каждая секция кода и как она взаимодействует с веб-службой, которую мы создадим.

## Создание Web API приложения

Как мы уже объяснили, приложение Web API - это обычное приложение MVC Framework, в котором есть специальный контроллер. В качестве примера мы создали новый проект MVC Framework под названием `WebServices` на шаблоне `Basic`. В следующих разделах мы будем добавлять все регулярные компоненты приложения MVC Framework - объекты моделей, хранилище, контроллеры и представления, а затем добавим контроллер API.

### *Подсказка*

*Существует шаблон `WebAPI`, который создает обычный проект MVC Framework и добавляет в него пару типичных контроллеров. Как вы уже поняли, мы предпочитаем шаблоны `Empty` и `Basic` и считаем, что другие варианты просто добавляют типичный код, который лучше не использовать.*

---

### Создаем модель и хранилище

В этом примере приложения мы создали особенно простую модель под названием `Reservation`, которую определили в файле `Reservation.cs` в папке `Models`. Определение этой модели показано в листинге 25-1 .

#### Листинг 25-1: Класс модели `Reservation`

```
using System.ComponentModel.DataAnnotations;

namespace WebServices.Models
{
    public class Reservation
    {
        public int ReservationId { get; set; }
        public string ClientName { get; set; }
        public string Location { get; set; }
    }
}
```

Свойство `ReservationId` однозначно идентифицирует каждый объект модели. Свойства `ClientName` и `Location` содержат значения данных модели. В этой главе нас не интересует значение этих свойств, потому что мы собираемся работать с Web API, так что мы выбрали нечто простое и универсальное.

В последних нескольких главах мы создавали приложения, в которых данные модели создавались внутри контроллера. Мы делали это потому, что были сосредоточены на других аспектах MVC Framework, не связанных с хранением данных модели. В этой главе мы собираемся создать интерфейс хранилища и его простую реализацию. На это стоит потратить время, потому что одна из главных характеристик контроллера API - это его простота, и мы хотим ее продемонстрировать.

Мы создали новый интерфейс в файле `IReservationRepository.cs` в папке `Models`, как показано в листинге 25-2.

**Листинг 25-2:** Интерфейс `IReservationRepository`

```
using System.Collections.Generic;

namespace WebServices.Models
{
    public interface IReservationRepository
    {
        IEnumerable<Reservation> GetAll();
        Reservation Get(int id);
        Reservation Add(Reservation item);
        void Remove(int id);
        bool Update(Reservation item);
    }
}
```

Это стандартный интерфейс хранилища; он определяет методы, которые позволяют извлекать (как индивидуально, так и в коллекции), добавлять, обновлять и удалять объекты моделей.

Мы также создали класс `ReservationRepository`, который определили в файле `ReservationRepository.cs` в папке `Models`. Этот класс реализует интерфейс `IReservationRepository` и определяет несколько объектов модели. Это не тот подход, который мы бы использовали в реальном приложении, но в данной главе он подходит для наших целей. Класс хранилища показан в листинге 25-3 .

**Листинг 25-3:** Класс `ReservationRepository`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

namespace WebServices.Models
{
    public class ReservationRepository : IReservationRepository
    {
        private List<Reservation> data = new List<Reservation>
        {
            new Reservation
            {
                ReservationId = 1,
                ClientName = "Adam",
                Location = "London"
            },
            new Reservation
            {
```

```

        ReservationId = 2,
        ClientName = "Steve",
        Location = "New York"
    },
    new Reservation
    {
        ReservationId = 3,
        ClientName = "Jacqui",
        Location = "Paris"
    },
};

private static ReservationRepository repo = new ReservationRepository();

public static IReservationRepository getRepository()
{
    return repo;
}

public IEnumerable<Reservation> GetAll()
{
    return data;
}

public Reservation Get(int id)
{
    var matches = data.Where(r => r.ReservationId == id);
    return matches.Count() > 0 ? matches.First() : null;
}

public Reservation Add(Reservation item)
{
    item.ReservationId = data.Count + 1;
    data.Add(item);
    return item;
}

public void Remove(int id)
{
    Reservation item = Get(id);
    if (item != null)
    {
        data.Remove(item);
    }
}

public bool Update(Reservation item)
{
    Reservation storedItem = Get(item.ReservationId);
    if (storedItem != null)
    {
        storedItem.ClientName = item.ClientName;
        storedItem.Location = item.Location;
        return true;
    }
    else
    {
        return false;
    }
}
}
}

```

Изменения в хранилище не сохраняются, так что при каждом запуске приложения в данных модель всегда будет три эталонных объекта.

## Создаем контроллер Home

В проекте можно свободно смешивать обычные контроллеры и контроллеры API. На самом деле, вам придется это делать, если вы хотите обеспечить поддержку HTML-клиентов в приложении, потому что контроллеры API будут возвращать только данные объекта, а не представления. Чтобы начать использовать приложение, мы создали контроллер `Home`, метод действия `Index` которого будет визуализировать представление по умолчанию. Мы не передаем в представление объекты модели, потому что хотим реализовать веб-службу и получать все необходимые данные от контроллера API. Контроллер `Home` показан в листинге 25-4 .

### Листинг 25-4: Контроллер Home в проекте WebServices

```
using System.Web.Mvc;

namespace WebServices.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }
    }
}
```

## Создаем представление и CSS

Мы добавили несколько стилей CSS в файл `/Content/Site.css` для элементов HTML, которые будут визуализированы после вызова действия `Index`. Эти дополнения показаны в листинге 25-5.

### Листинг 25-5: Дополнения в файле Site.css

```
table { margin: 10px 0; }
th { text-align: left; }

.nameCol { width: 100px; }
.locationCol { width: 100px; }
.selectCol { width: 30px; }
.display { float: left; border: thin solid black; margin: 10px; padding: 10px; }
.display label { display: inline-block; width: 100px; }
```

Действие `Index` контроллера `Home` визуализирует представление `/Views/Home/Index.cshtml`, которое показано в листинге 25-6. Здесь мы не установили значения для свойства `Layout`, что означает, что будет использоваться макет по умолчанию. Следовательно, наши стили CSS и библиотеки jQuery (которые мы будем использовать позже) будут загружены браузером автоматически при визуализации представления. В секцию `scripts` мы добавили элемент `script` для библиотеки ненавязчивого Ajax (которую мы также будем использовать позже).

### Листинг 25-6: Index.cshtml

```
@{
    ViewBag.Title = "Index";
}

@section scripts {
    <script src="~/Scripts/jquery.unobtrusive-ajax.js"></script>
```

```

}

<div id="summaryDisplay" class="display">
    <h4>Reservations</h4>
    <table>
        <thead>
            <tr>
                <th class="selectCol"></th>
                <th class="nameCol">Name</th>
                <th class="locationCol">Location</th>
            </tr>
        </thead>
        <tbody id="tableBody">
            <tr><td colspan="3">The data is loading</td></tr>
        </tbody>
    </table>
    <div id="buttonContainer">
        <button id="refresh">Refresh</button>
        <button id="add">Add</button>
        <button id="edit">Edit</button>
        <button id="delete">Delete</button>
    </div>
</div>

<div id="addDisplay" class="display">
    <h4>Add New Reservation</h4>
    @{
        AjaxOptions addAjaxOpts = new AjaxOptions
        {
            // options will go here
        };
    }
    @using (Ajax.BeginForm(addAjaxOpts))
    {
        @Html.Hidden("ReservationId", 0)
        <p><label>Name:</label>@Html.Editor("ClientName")</p>
        <p><label>Location:</label>@Html.Editor("Location")</p>
        <button type="submit">Submit</button>
    }
</div>

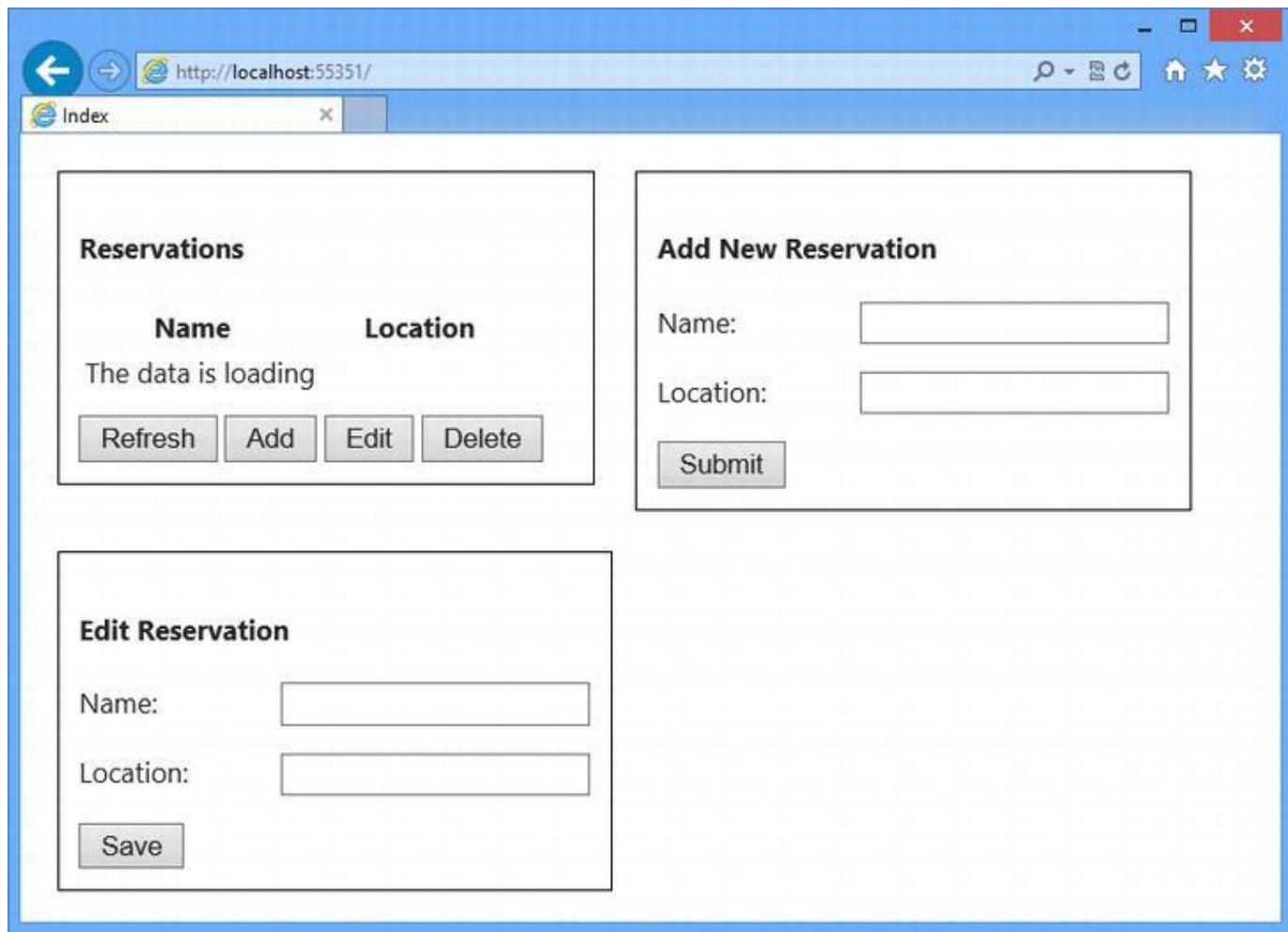
<div id="editDisplay" class="display">
    <h4>Edit Reservation</h4>
    <form id="editForm">
        <input id="editReservationId" type="hidden" name="ReservationId" />
        <p><label>Name:</label><input id="editClientName" name="ClientName" /></p>
        <p><label>Location:</label><input id="editLocation" name="Location" /></p>
    </form>
    <button id="submitEdit" type="submit">Save</button>
</div>

```

HTML, который генерирует это представление, разбит на три секции. Каждая секция определена в элементе `div` с классом `display`, и на данный момент пользователю отображаются все три (мы изменим видимость этих элементов, когда мы будем использовать код JavaScript далее в этой главе).

Чтобы увидеть, как HTML отображается в браузере, запустите приложение. Ни одна из кнопок не будет работать, и данные не будут отображаться, но вы можете получить представление о структуре и макете, как показано на рисунке 25-1 .

Рисунок 25-1: HTML, сгенерированный представлением Index.cshtml



Когда мы закончим работу над приложением, пользователю будет показана только секция Reservations. Мы загрузим данные с сервера и заполним ими таблицу.

Секция Add New Reservation содержит форму, которая использует ненавязчивый Ajax для отправки данных к серверу и создания новых объектов Reservation в хранилище. В объекте AjaxOptions, с помощью которого мы будем настраивать запрос Ajax, пока еще нет опций, но мы вернемся и создадим их, когда остальные части приложения будут готовы.

Секция Edit Reservation позволит пользователю изменять существующий в хранилище объект Reservation. В этой секции мы не использовали форму с ненавязчивым Ajax, потому что для разнообразия хотим использовать поддержку Ajax из библиотеки jQuery напрямую (что в любом случае делает библиотека ненавязчивого Ajax).

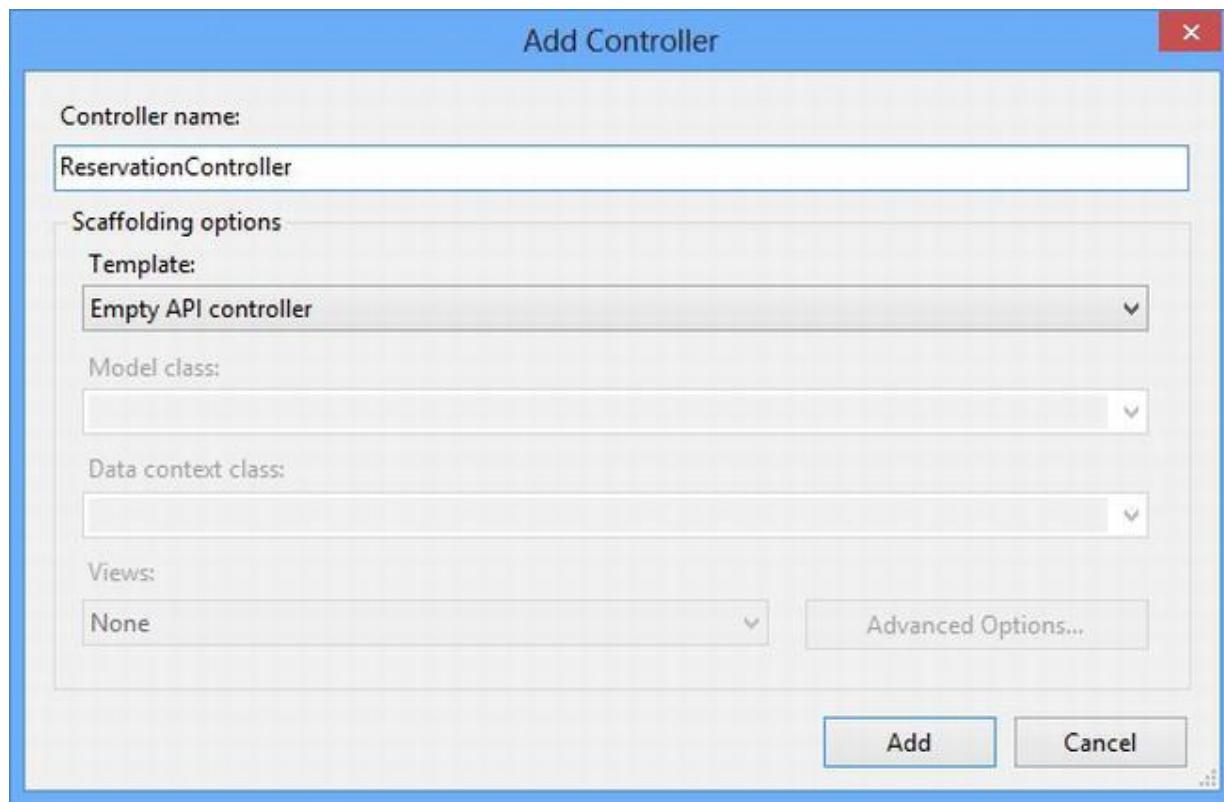
## Создание API контроллера

Теперь мы готовы определить Web API, который будем использовать в коде JavaScript для взаимодействия с содержимым хранилища. В этом разделе мы добавим в проект API Controller и объясним, как он работает.

Для создания контроллера кликните правой кнопкой мыши по папке Controllers в окне Solution Explorer и выберите Add - Controller из контекстного меню. В поле Controller name введите

ReservationController и выберите Empty API controller из выпадающего списка Template, как показано на рисунке 25-2.

**Рисунок 25-2:** Добавляем новый контроллер API в проект



Опция Empty API controller создает контроллер API без методов действий. В выпадающем списке доступны несколько других опций, которые будут создавать контроллеры API с шаблонными методами действий, которые можно заполнить. Но мы предпочитаем начинать с пустого класса и определять функции, которые нам нужны. Начальный класс контроллера, созданный Visual Studio, показан в листинге 25-7.

**Листинг 25-7:** Контроллер, который создает опция Empty API Controller

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Http;

namespace WebServices.Controllers
{
    public class ReservationController : ApiController
    {
    }
}
```

Мы выделили базовый класс контроллера Reservation, потому что в нем заметны отличия от обычного контроллера. System.Web.Http.ApiController реализует интерфейс IController, описанный в главе 15, но совершенно иначе обрабатывает запросы, чем обычный класс System.Web.Mvc.Controller, который был основой для всех других примеров в этой книге (за исключением того, где мы создали собственную реализацию IController). В листинге 25-8

показано, как мы добавили в контроллер Reservation хранилище и определили методы действий Web API, которые потребуются для доступа к объектам Reservation.

#### Листинг 25-8: Завершаем контроллер API Reservation

```
using System.Collections.Generic;
using System.Web.Http;
using WebServices.Models;

namespace WebServices.Controllers
{
    public class ReservationController : ApiController
    {
        private IReservationRepository repo = ReservationRepository.getRepository();

        public IEnumerable<Reservation> GetAllReservations()
        {
            return repo.GetAll();
        }

        public Reservation GetReservation(int id)
        {
            return repo.Get(id);
        }

        public Reservation PostReservation(Reservation item)
        {
            return repo.Add(item);
        }

        public bool PutReservation(Reservation item)
        {
            return repo.Update(item);
        }

        public void DeleteReservation(int id)
        {
            repo.Remove(id);
        }
    }
}
```

Это все, что требуется для создания Web API. Мы создали экземпляр нашего класса хранилища и пять методов действий, которые обеспечивают доступ к объектам модели Reservation, которые содержаться в хранилище.

#### Подсказка

*Обратите внимание, что для получения IReservationRepository мы вызываем статический метод ReservationRepository.getRepository. Для обслуживания запросов к API создаются новые экземпляры контроллеров, и, если бы нам нужно было создать новый объект ReservationRepository в самом контроллере, наши изменения никак не отразились бы на данных, отправляемых клиенту. Это особенность нашего хранилища «в памяти» - такого не будет происходить при работе с реальными хранилищами, которые сохраняют данные в базах.*

## Тестируем контроллер API

Мы вскоре расскажем, как работает контроллер API, но перед этим выполним простой тест. Запустите приложение. Когда браузер загрузит корневой URL проекта, перейдите по ссылке `/api/reservation`. Результат, который вы увидите, будет зависеть от вашего браузера. Если вы используете Internet Explorer 10, то вам будет предложено сохранить или открыть файл, который содержит следующие данные JSON:

```
[{"ReservationId":1,"ClientName":"Adam","Location":"London"},  
 {"ReservationId":2,"ClientName":"Steve","Location":"New York"},  
 {"ReservationId":3,"ClientName":"Jacqui","Location":"Paris"}]
```

Если перейти по той же ссылке в другом браузере, например, Google Chrome или Mozilla Firefox, то браузер отобразит следующие данные XML:

```
<ArrayOfReservation>  
  <Reservation>  
    <ClientName>Adam</ClientName>  
    <Location>London</Location>  
    <ReservationId>1</ReservationId>  
  </Reservation>  
  <Reservation>  
    <ClientName>Steve</ClientName>  
    <Location>New York</Location>  
    <ReservationId>2</ReservationId>  
  </Reservation>  
  <Reservation>  
    <ClientName>Jacqui</ClientName>  
    <Location>Paris</Location>  
    <ReservationId>3</ReservationId>  
  </Reservation>  
</ArrayOfReservation>
```

Здесь можно отметить несколько интересных моментов. Во-первых, наш запрос к ссылке `/api/reservation` вернул список всех наших объектов модели и их свойств, из чего мы делаем вывод, что был вызван метод действия `GetAllReservations` в контроллере `Reservation`.

Во-вторых, разные браузеры получили разные форматы данных. Если вы сами попробуете перейти по ссылке из разных браузеров, то получите разные результаты, потому что более поздние версии браузеров могут иначе отправлять запросы; в любом случае вы увидите, что один запрос вернулся JSON, другой - XML. (Вы также можете понять, почему JSON в значительной степени заменил XML в веб-службах. XML использует более подробную разметку и его труднее обрабатывать, особенно если вы используете JavaScript).

Различные форматы данных используются потому, что Web API использует заголовок HTTP `Accept`, содержащийся в запросе, чтобы узнать, с каким типом данных клиент предпочел бы работать. Internet Explorer получает JSON, потому что он посыпает следующий заголовок:

```
Accept: text/html, application/xhtml+xml, */*
```

Браузер указал, что он прежде всего он предпочитает работать с форматом `text/html`, далее - `application/xhtml+xml`. В конце заголовка указано `*/*`, что означает, что браузер примет любой тип данных, если первые два недоступны.

Web API поддерживает JSON и XML, но отдает предпочтение JSON, который и использует в ответе, опираясь на часть заголовка `Accept */*`. Вот заголовок `Accept`, который отправляет Google Chrome:

Accept:text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8

Здесь мы выделили важную деталь: Chrome сообщил, что он предпочитает получать данные application/xml всем остальным. Web API учитывает это предпочтение и отправляет данные XML.

Мы останавливаемся на этом потому, что общая проблема при работе с Web API - получение нежелательного формата данных. Так происходит потому, что в заголовке Accept указывается неправильное предпочтение формата, или оно вообще отсутствует.

## Как работают API контроллеры

Чтобы больше узнать о том, как работает контроллер API, перейдите по ссылке /api/reservation/3. Вы увидите следующий JSON (или соответствующий XML, если вы используете другой браузер):

```
{"ReservationId":3,"ClientName":"Jacqui","Location":"Paris"}
```

На этот раз наш запрос вернул информацию об объекте Reservation, значение ReservationId которого соответствует последнему сегменту URL. Формат URL и использование сегментов URL должны напомнить вам о главе 13, в которой объяснялись принципы работы маршрутов в MVC Framework.

В контроллерах API есть своя конфигурация маршрутизации, которая полностью отделена от всего остального приложения. Конфигурацию по умолчанию, которую Visual Studio создает для новых проектов, можно увидеть в файле /App\_Start/WebApiConfig.cs, который показан в листинге 25-9 .

**Листинг 25-9:** Содержимое файла WebApiConfig.cs по умолчанию

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.Http;

namespace WebServices
{
    public static class WebApiConfig
    {
        public static void Register(HttpConfiguration config)
        {
            config.Routes.MapHttpRoute(
                name: "DefaultApi",
                routeTemplate: "api/{controller}/{id}",
                defaults: new { id = RouteParameter.Optional }
            );
        }
    }
}
```

Файл WebApiConfig.cs содержит маршруты, которые используются контроллерами API, но обычные маршруты MVC, которые определены в файле RouteConfig.cs, в нем не используются. Функция Web API реализована как автономная функция ASP.NET, ее можно использовать вне MVC Framework, следовательно, Microsoft дублировала ключевую функциональность MVC Framework из пространства имен System.Web.Http, чтобы отделить Web API от MVC. (Это кажется странным при разработке приложения в MVC Framework, но все же имеет смысл, поскольку Microsoft позволяет использовать Web API разработчикам, которые не работают с MVC).

При запуске приложения вызывается статический метод `Register` из метода `Application_Start` файла `Config.asax`; этот метод используется для регистрации маршрутов Web API. Метод `Register` получает объект `HttpConfiguration`, который предоставляет доступ к маршрутизации через свойство `Routes`. Маршруты создаются с помощью метода `MapHttpRoute`.

## Принцип выбора действий в контроллере API

Стандартный маршрут Web API, который показан в листинге 25-9, содержит статический сегмент `api` и переменные сегментов `controller` и `id`, причем последний является дополнительным. Ключевым отличием от обычного маршрута MVC является то, что в нем нет переменной сегмента `action`; с этого момента поведение контроллеров API будет более понятным.

Когда в приложение поступает запрос, соответствующий маршруту Web API, действие определяется по методу HTTP, который использовался в запросе. Когда мы проверили работу контроллера API, перейдя по ссылке `/api/reservation`, браузер указал метод GET.

Класс `ApiController`, который является основой контроллеров API, узнает из маршрута, какой контроллер должен обрабатывать запрос, и использует метод HTTP для поиска подходящих методов действий. По соглашению методы действий контроллеров API должны содержать приставку с именем метода действия, который они поддерживают, и тип модели, с которым они работают. Но это только соглашение, поскольку Web API найдет любой метод действия, в имени которого указан метод HTTP, использовавшийся в запросе.

В нашем примере это означает, что для запроса GET выбор будет проводиться между `GetAllReservations` и `GetReservation`, но если бы методы назывались `DoGetReservation` или просто `ThisIsTheGetAction`, они бы тоже учитывались.

Чтобы выбрать между двумя методами действий, контроллер смотрит на аргументы, которые принимают контроллеры, и переменные маршрутизации. При запросе API по ссылке `/api/reservation`, в которой нет переменных маршрутизации за исключением `controller`, был выбран метод `GetAllReservations`, потому что он не принимает аргументов. Когда мы запросили URL `/api/reservation/3`, мы поставили значение для опциональной переменной сегмента `id`; для этого запроса лучше подходит метод `GetReservation`, потому что он принимает аргумент `id`.

Для вызова других действий в нашем контроллере API `Reservation` будут использоваться другие методы HTTP: POST, DELETE и PUT. Это служит основой стиля REST (Representation State Transfer) Web API, также известного как REST-сервис, в котором операция определяется комбинацией URL и метода HTTP, использованного в запросе к данному URL.

### Примечание

*REST является стилем API, а не четко определенной спецификацией, поэтому существуют разногласия по поводу того, что именно делает веб-службу REST-сервисом. Одной из точек разногласий является то, что «туристы» не считают веб-службы, которые возвращают JSON, REST-сервисами. Как и все споры по поводу архитектурных шаблонов, этот возник из-за скучных и несерьезных причин. Мы рассматриваем применение шаблонов с практической стороны и убеждены, что сервисы JSON являются REST-сервисами.*

## Соотнесение методов HTTP с методами действий

Как мы уже объяснили, базовый класс `ApiController` использует метод HTTP, чтобы узнать, какой метод действий должен обрабатывать запрос. Это хороший подход, но он означает, что вам придется назначать методам действий неестественные имена, противоречащие соглашениям, которые вы могли бы использовать в приложении. Например, метод `PutReservation` более естественно было бы назвать `UpdateReservation`. Имя `UpdateReservation` не только сделало бы цель метода более очевидной, но и позволило бы напрямую соотносить действия контроллера и методы хранилища.

### Подсказка

*Вы могли бы попытаться наследовать класс хранилища от `ApiController` и использовать методы хранилища напрямую как Web API. Мы настоятельно не рекомендуем так делать и советуем создавать отдельный контроллер, даже если он будет таким же простым, как в нашем примере. В какой-то момент методы, которые вы захотите использовать в API, и возможности хранилища начнут отличаться, и отдельный контроллер API упростит решение этой проблемы.*

Пространство имен `System.Web.Http` содержит набор атрибутов, с помощью которых можно указать, для какого метода HTTP должно использоваться действие. В листинге 25-10 показано, как мы применили два из этих атрибутов, чтобы создать более естественные имена методов.

**Листинг 25-10:** Указываем методы HTTP, которые поддерживает действие, с помощью атрибутов

```
using System.Collections.Generic;
using System.Web.Http;
using WebServices.Models;

namespace WebServices.Controllers
{
    public class ReservationController : ApiController
    {
        private IReservationRepository repo = ReservationRepository.getRepository();

        public IEnumerable<Reservation> GetAllReservations()
        {
            return repo.GetAll();
        }

        public Reservation GetReservation(int id)
        {
            return repo.Get(id);
        }

        [HttpPost]
        public Reservation CreateReservation(Reservation item)
        {
            return repo.Add(item);
        }

        [HttpPut]
        public bool UpdateReservation(Reservation item)
        {
            return repo.Update(item);
        }

        public void DeleteReservation(int id)
        {
            repo.Remove(id);
        }
    }
}
```

```
    }
}
```

Вы можете заметить, что в Web API дублируются функции MVC Framework, которые мы рассмотрели в главе 17. Атрибуты `HttpPost` и `HttpPut`, которые мы использовали в листинге 25-10, имеют точно такую же цель, что и одноименные атрибуты в главе 17, но они определены в пространстве имен `System.Web.Http`, а не `System.Web.Mvc`. Эти атрибуты работают таким же образом, и в итоге мы получили более полезные имена методов, которые будут работать с HTTP-методами `POST` и `PUT`. (Конечно, существуют атрибуты для всех методов HTTP, в том числе `GET`, `DELETE` и т.д.)

## Написание JavaScript кода для использования Web API

Мы создали наш контроллер API и объяснили, как URL вида `/api/reservation/3` сопоставляется с методом действия в зависимости от метода HTTP. Теперь пора написать код JavaScript, с помощью которого мы будем использовать созданный Web API. Мы будем использовать jQuery, чтобы управлять HTML-элементами представления `Views/Home/Index.cshtml` и обрабатывать запросы Ajax, которые будем отправлять к действиям контроллера `Reservation`.

jQuery - это отличная многофункциональная библиотека JavaScript, которую мы часто используем в собственных проектах и рекомендуем к использованию вам. В этой книге мы не сможем поместить руководство по jQuery, поэтому будем создавать функциональность JavaScript и рассказывать вам, что делает каждый блок кода, не вдаваясь в подробности о том, как работают функции jQuery. Дополнительные сведения о jQuery можно найти на сайте [jquery.com](http://jquery.com) или в книге Адама Pro jQuery.

### Создаем базовую функциональность

Для начала мы создадим папку `/Scripts/Home` и добавим в нее новый файл JavaScript под названием `Index.js` (как упоминалось в главе 24, мы организовываем скрипты для отдельных приложений, следя соглашению). Прежде чем сделать что-нибудь еще, мы добавим элемент `script` к определению секции `scripts` в представлении `/Views/Home/Index.cshtml`, который будет загружать наш код JavaScript, как показано в листинге 25-11.

**Листинг 25-11:** Добавляем элемент `script` для файла `Index.js` в представление `Index.cshtml`

```
@{
    ViewBag.Title = "Index";
}

@section scripts {
    <script src="~/Scripts/jquery.unobtrusive-ajax.js"></script>
    <script src="~/Scripts/Home/Index.js"></script>
}
```

Далее мы добавим в файл `Index.js` необходимые базовые функции, как показано в листинге 25-12.

**Листинг 25-12:** Добавляем базовые функции в файл `Index.js`

```
function selectView(view) {
    $('.display').not('#' + view + "Display").hide();
    $('#' + view + "Display").show();
}
function getData() {
```

```

$.ajax({
  type: "GET",
  url: "/api/reservation",
  success: function (data) {
    $('#tableBody').empty();
    for (var i = 0; i < data.length; i++) {
      $('#tableBody').append('<tr><td><input id="id" name="id" type="radio"' +
        + 'value="' + data[i].ReservationId + '" /></td>' +
        + '<td>' + data[i].ClientName + '</td>' +
        + '<td>' + data[i].Location + '</td></tr>');
    }
    $('input:radio')[0].checked = "checked";
    selectView("summary");
  }
});
$(document).ready(function () {
  selectView("summary");
  getData();
  $("button").click(function (e) {
    var selectedRadio = $('input:radio:checked')
    switch (e.target.id) {
      case "refresh":
        getData();
        break;
      case "delete":
        break;
      case "add":
        selectView("add");
        break;
      case "edit":
        selectView("edit");
        break;
      case "submitEdit":
        break;
    }
  });
});

```

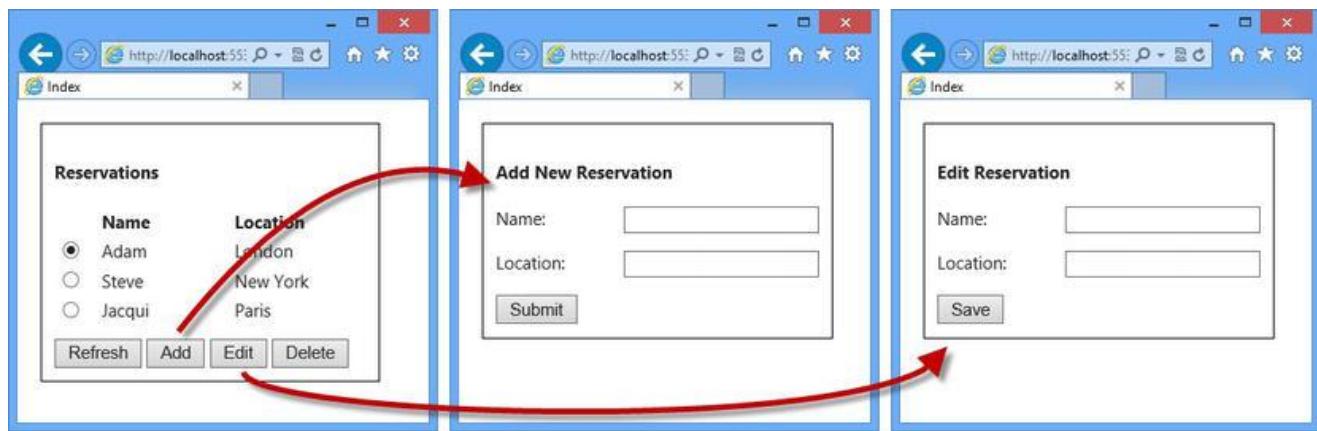
Мы определили три функции. Первая, `selectView`, изменяет видимость элементов `div` в классе `display`, так что отображаться будет только один набор элементов. Вторая функция, `GetData`, использует поддержку Ajax в jQuery для отправки запросов по адресу `/api/reservation`. Для добавления строк в таблицу в представлении используется массив объектов JSON; он заменяет заполнитель `The data is loading`, который был виден на рисунке 25-1. Каждая строка в таблице содержит переключатель, с помощью которого пользователь сможет отредактировать или удалить объект `Reservation`.

Последняя функция будет передана в функцию `jQuery ready`, что означает, что она не будет выполнена до окончания загрузки и обработки содержимого страницы браузером. Мы вызываем функцию `selectView` для отображения только содержимого элемента `summaryDisplay`, `getData` - для загрузки данных по ссылке `/api/reservation` (как мы продемонстрировали ранее, это приведет к вызову метода `GetAllReservations` в контроллере `Reservation`). Мы также настроили обработчик событий, который будет выполняться после нажатия любой кнопки на странице. Мы использовали оператор `switch`, чтобы различать элементы `button` в зависимости от значения атрибута `id`. В его операторе `case` мы создаем различные запросы, которые затем отправим на сервер.

В данный момент для кнопки `Refresh` мы вызываем функцию `getData`, которая перезагружает данные от сервера, для кнопок `Edit` и `Add` - функцию `selectView`, которая отображает элементы, необходимые для создания и редактирования объектов модели.

Если вы запустите приложение и перейдете по корневой ссылке, то увидите изменения, созданные нашим базовым кодом JavaScript, как показано на рисунке 25-3.

**Рисунок 25-3:** Результат выполнения JavaScript кода



### Добавляем поддержку редактирования новых объектов Reservation

Мы хотим использовать все методы действий контроллера `Reservation`, поэтому наш подход к редактированию объектов `Reservation` будет немного странным. В документе HTML у нас уже есть все необходимые данные для редактирования объекта `Reservation`, но мы запросим у сервера один объект `Reservation`, чтобы использовать объект `GetReservation`. В листинге 25-13 показано, как мы добавили операторы в файл `Index.js` для ответа на нажатие кнопки `Edit`.

**Листинг 25-13:** Отвечааем на нажатие кнопки `Edit`

```
$(document).ready(function() {
    selectView("summary");
    getData();
    $("button").click(function(e) {
        var selectedRadio = $('input:radio:checked')
        switch (e.target.id) {
            case "refresh":
                getData();
                break;
            case "delete":
                break;
            case "add":
                selectView("add");
                break;
            case "edit":
                $.ajax({
                    type: "GET",
                    url: "/api/reservation/" + selectedRadio.attr('value'),
                    success: function(data) {
                        $('#editReservationId').val(data.ReservationId);
                        $('#editClientName').val(data.ClientName);
                        $('#editLocation').val(data.Location);
                        selectView("edit");
                    }
                });
                break;
            case "submitEdit":
                break;
        }
    });
});
```

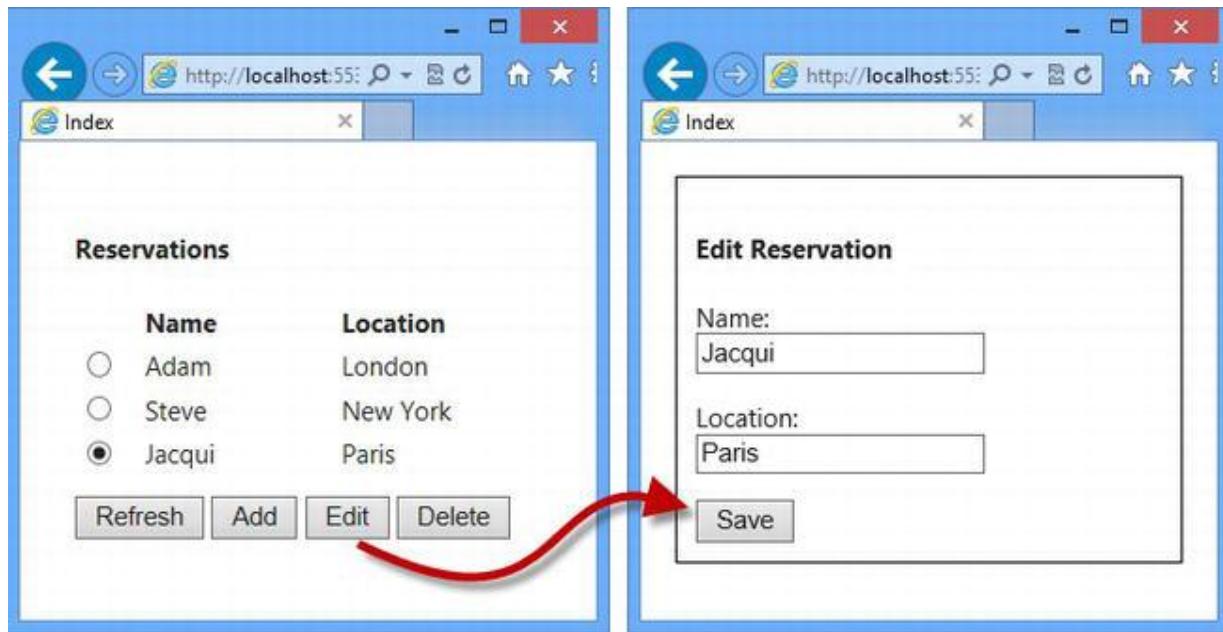
При создании строк таблицы в функции `getData` мы использовали значение свойства `ReservationId` каждого объекта `Reservation`, чтобы установить значение для элемента `radio button`, например:

```
<input name="id" id="id" type="radio" value="3"/>
```

Когда пользователь нажимает кнопку `Edit`, мы находим выбранный переключатель и используем его атрибут `value` в URL, который запросим у сервера. Если пользователь выбрал радиокнопку, показанную ранее, то мы запросим URL `/api/reservation/3`. Мы сообщаем jQuery, что хотим отправить запрос GET, и сочетание URL и метода HTTP приводят нас к методу действия `GetReservation` контроллера `Reservation`.

Мы используем полученные данные JSON, чтобы установить значения элементов `input` в секции `editDisplay`, а затем вызываем функцию `selectView`, чтобы отобразить их пользователю, как показано на рисунке 25-4.

**Рисунок 25-4:** Выбор объекта для редактирования



Чтобы разрешить пользователю сохранить изменения, нам нужно заполнить блок `case`, который работает с `id` кнопки `submitEdit`, как показано в листинге 25-14.

**Листинг 25-14:** Сохраняем изменения на сервере

```
$(document).ready(function () {
    selectView("summary");
    getData();
    $("button").click(function (e) {
        var selectedRadio = $('input:radio:checked')
        switch (e.target.id) {
            case "refresh":
                getData();
                break;
            case "delete":
                break;
            case "add":
                selectView("add");
                break;
            case "edit":
                $.ajax({
```

```

        type: "GET",
        url: "/api/reservation/" + selectedRadio.attr('value'),
        success: function (data) {
            $('#editReservationId').val(data.ReservationId);
            $('#editClientName').val(data.ClientName);
            $('#editLocation').val(data.Location);
            selectView("edit");
        }
    });
    break;
case "submitEdit":
    $.ajax({
        type: "PUT",
        url: "/api/reservation/" + selectedRadio.attr('value'),
        data: $('#editForm').serialize(),
        success: function (result) {
            if (result) {
                var cells = selectedRadio.closest('tr').children();
                cells[1].innerText = $('#editClientName').val();
                cells[2].innerText = $('#editLocation').val();
                selectView("summary");
            }
        }
    });
    break;
}
});

```

Мы используем те же URL, что и для получения объекта `Reservation`, `/api/reservation/3`, но с методом HTTP `PUT`, следовательно, для обработки запроса будет использоваться метод действия `PutReservation` контроллера `Reservation`. Напомним, мы определили его следующим образом:

```

public bool PutReservation(Reservation item) {
    return repo.Update(item);
}

```

Обратите внимание, что аргументом этого метода действия является объект `Reservation`. Контроллеры API используют те же механизмы связывания, которые мы описали в главе 22, что означает, что нам не придется преобразовывать данные запроса в объект модели.

## Добавляем поддержку удаления объектов `Reservation`

Вы уже поняли принцип работы API, и как метод HTTP изменяет метод действия, который будет обрабатывать наш запрос, даже если мы отправляем его по той же ссылке. В листинге 25-15 показано, как мы добавили поддержку удаления объектов `Reservation`, для чего используется метод HTTP `DELETE`.

**Листинг 25-15:** Добавляем поддержку удаления объектов `Reservation`

```

case "delete":
    $.ajax({
        type: "DELETE",
        url: "/api/reservation/" + selectedRadio.attr('value'),
        success: function (data) {
            selectedRadio.closest('tr').remove();
        }
    });
    break;
}

```

В элементе `table` мы удаляем строку, которая содержит данные объекта `Reservation`, который был удален. Мы делаем это независимо от результата, который получим от сервера, что не будет иметь смысла в реальном проекте.

## Добавляем поддержку создания объектов `Reservation`

Для создания новых объектов `Reservation` мы применим несколько иной подход. Для создания запросов `PUT` или `DELETE` легче будет использовать поддержку Ajax в `jQuery`, но для запросов `POST` и `GET` можно без проблем использовать ненавязчивый Ajax. Чтобы добавить поддержку создания новых объектов данных, нам понадобится только настроить объект `AjaxOptions`, который мы используем во вспомогательном методе `Ajax.BeginForm` в представлении `Index.cshtml`, как показано в листинге 25-16.

### Подсказка

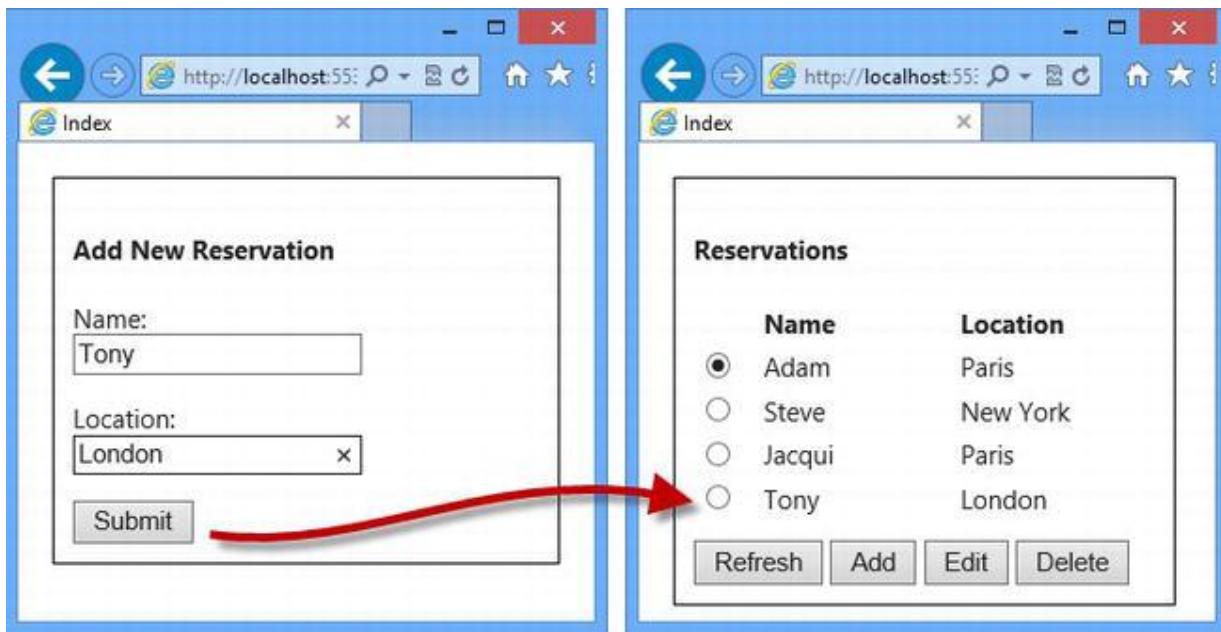
*Если вы хотите использовать формы Ajax для всех запросов, или вы хотите использовать REST-сервис в браузере, который поддерживает только методы `GET` и `POST`, то с помощью вспомогательного метода `Html.HttpMethodOverride` вы можете добавить в форму скрытый элемент, который будет интерпретирован контроллером API и использован для обращения к методам действий. Переопределять можно только запросы `POST`, но эта резервная техника может быть полезной, особенно для старых браузеров.*

**Листинг 25-16:** Настраиваем объект `AjaxOptions` для создания новых объектов модели

```
<div id="addDisplay" class="display">
    <h4>Add New Reservation</h4>
    @{
        AjaxOptions addAjaxOpts = new AjaxOptions
        {
            OnSuccess = "getData",
            Url = "/api/reservation"
        };
    }
    @using (Ajax.BeginForm(addAjaxOpts))
    {
        @Html.Hidden("ReservationId", 0)
        <p><label>Name:</label>@Html.Editor("ClientName")</p>
        <p><label>Location:</label>@Html.Editor("Location")</p>
        <button type="submit">Submit</button>
    }
</div>
```

Эта форма будет отправлена по умолчанию с помощью метода `POST`, и нам не нужно создавать URL динамически, потому что метод действия `PostReservation` не принимает переменные сегментов в качестве параметров (он принимает объект `Reservation`, который создается механизмом связывания). Когда пользователь отправляет форму на сервер, будет вызван метод действия `PostReservation`, который создаст в хранилище новый объект `Reservation`. Когда запрос завершен, мы вызываем метод `getData`, чтобы обновить данные клиента и отобразить итоговое представление. Для простоты кода `JavaScript` мы этого не делаем, хотя сервер и отправляет нам вновь созданный объект в формате `JSON`, с помощью которого мы могли бы добавить в таблицу новую строку. Результат создания нового объекта `Reservation` показан на рисунке 25-5.

Рисунок 25-5: Добавляем новый объект Reservation



И это весь код, необходимый для завершения нашего Web API и простого приложения, в котором он используется. Как мы заметили ранее в этой главе, функция Web API очень проста, и все время уйдет на создание и тестирование клиента, который его использует. Чтобы создать сам API, необходимо наследовать новый контроллер от `ApiController` и создать методы действий с теми же именами, что и методы HTTP, с помощью которых вы хотите к ним обращаться.

## Резюме

В этой главе мы показали вам, как с помощью функции Web API создать REST-сервис, который смогут использовать HTTP-клиенты. Хотя на самом деле Web API не является частью MVC Framework, он очень близок по принципам и структуре к MVC и, таким образом, знаком разработчикам MVC. Как мы показали, контроллеры Web API можно добавить в приложение наряду с обычными контроллерами MVC. В главе 26 (в заключение этой книги) мы покажем вам, как развертывать приложения MVC .

# Развертывание приложения

В этой главе мы покажем вам, как подготовить приложение к развертыванию и провести развертывание. Существует много различных способов развертывания приложений MVC Framework, есть также много целевых серверов. Вы можете развернуть приложение на машине под управлением Windows Server с запущенными службами IIS, которые вы запускаете и конфигурируете локально; возможно развертывание на удаленном хостинге, который будет конфигурировать службы для вас; все чаще встречается развертывание на облачной инфраструктуре, которая обеспечит все необходимые для вашего приложения ресурсы.

Нам сложно было решить, как создать полезный пример развертывания для этой главы. Мы исключили развертывание напрямую в IIS, потому что конфигурация сервера - процесс долгий и сложный, и большинство разработчиков MVC, которые работают с локальными серверами, оставляют конфигурацию и развертывание IT-группе. Мы также исключили развертывание на управляемом хостинге, потому что каждая компания-провайдер определяет свой процесс развертывания и никто не устанавливает стандарты для хостингов.

Таким образом, все, что нам осталось, - это развертывание на Windows Azure, облачной платформе Microsoft и с хорошей поддержкой приложений MVC. Мы не утверждаем, что Azure подходит для всех вариантов развертывания, но нам нравится, как она работает; используя ее в этой главе, мы сосредоточимся на процессе развертывания, а не будем отвлекаться на вопросы конфигурации. Во время написания этой главы была доступна бесплатная 90-дневная пробная версия Azure (некоторые подписки MSDN тоже включают Azure); следовательно, вы сможете повторить пример этой главы, даже если вы не собираетесь использовать Azure для развертывания своих приложений. Для начала мы покажем вам, как подготовить приложение к развертыванию, и затем перейдем собственно к развертыванию.

## *Внимание!*

*Мы рекомендуем вам потренироваться с тестовым приложением и сервером, прежде чем развернуть реальное приложение в производственной среде. Как и любой другой аспект жизненного цикла разработки программного обеспечения, процесс развертывания только выигрывает от тестирования. Мы знаем жуткие истории о командах разработчиков, которые уничтожили рабочие приложения из-за чрезмерноспешной или плохо протестированной процедуры развертывания. Нельзя сказать, что функции развертывания ASP.NET особенно опасны - нет, но любое взаимодействие, которое подразумевает запуск приложения с реальными пользовательскими данными, необходимо тщательно продумать и спланировать.*

---

Развертывание приложения было утомительным и подверженным ошибкам процессом, но Microsoft не пожалела усилий для улучшения средств развертывания в Visual Studio, так что даже если вам необходимо развернуть приложение на другой инфраструктуре, вы убедитесь, что Visual Studio выполнит за вас много тяжелой работы.

# Подготовка приложения для публикации

Мы собираемся развернуть приложение SportsStore, которое создали в первой части этой книги. Мы выбрали SportsStore, потому что оно использует базу данных, а настройка и проверка работы базы данных является важной частью любого процесса развертывания.

### Подсказка

*Когда мы говорим о развертывании приложения SportsStore, мы имеем в виду проект SportsStore.WebUI. Нам не нужно отдельно развертывать два других проекта, которые мы создали вместе с ним. Вывод проекта SportsStore.Domain включен в проект WebUI, а модульные тесты из проекта SportsStore.UnitTests не потребуются в рабочем приложении.*

## Выявляем ошибки представлений

Как мы уже объяснили в главе 18, представления Razor компилируются сервером при запуске приложения, а не когда мы создаем проект в Visual Studio. Как правило, единственный способ обнаружить ошибки компилятора - это систематически обращаться к каждому действию и визуализировать каждое представление. Это утомительная и не всегда успешная техника, особенно если в зависимости от состояния модели визуализируются разные представления.

В проекте можно активировать специальную опцию, которая будет компилировать наши представления и сообщать об ошибках компиляции. Эту функцию нельзя активировать из Visual Studio; откройте файл SportsStore/WebUI/SportsStore.WebUI.csproj в блокноте. Найдите элемент MvcBuildViews и измените его значение на true, как показано в листинге 26-1.

### Листинг 26-1: Активируем опцию MvcBuildViews

```
...
<PropertyGroup>
    ... other settings omitted ...
    <MvcBuildViews>true</MvcBuildViews>
    ... other settings omitted ...
</PropertyGroup>
...
```

Сохраните изменения и вернитесь в Visual Studio, которая предложит вам перезагрузить проект. Теперь, когда вы скомпилируете проект, Visual Studio скомпилирует файлы представлений и сообщит об ошибках. Чтобы продемонстрировать, как это работает, мы добавили ошибку в представление /Views/Product/List.cshtml, как показано в листинге 26-2.

### Листинг 26-2: Добавляем ошибку в файл представления

```
@model SportsStore.WebUI.Models.ProductsListViewModel
 @{
    ViewBag.Title = "Products";
}

@ModelError.NotARealProperty

@foreach (var p in Model.Products)
{
    Html.RenderPartial("ProductSummary", p);
}



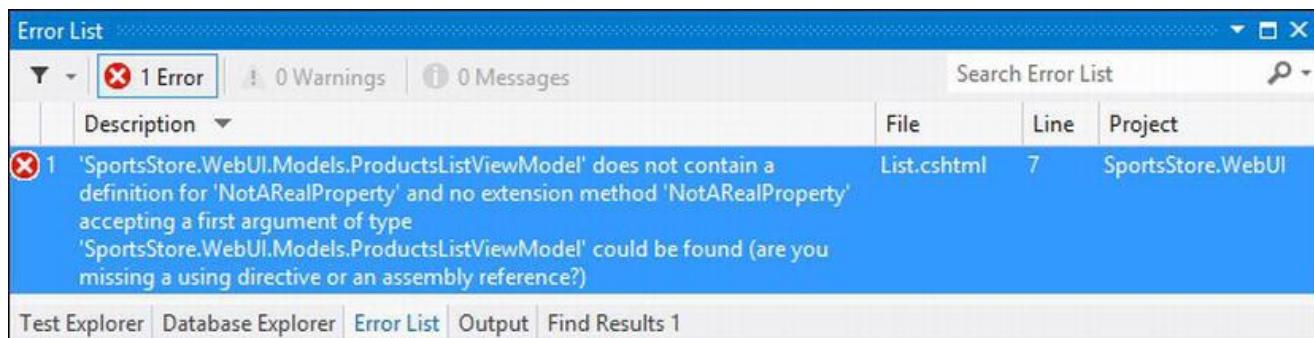
@Html.PageLinks(Model.PagingInfo,
        x => Url.Action("List", new { page = x, category = Model.CurrentCategory }))


```

Как правило, наша попытка визуализировать свойство не будет отображаться, пока мы не запустим приложение и перейдем по ссылке, которая визуализирует представление List.cshtml. Но, когда

опция MvcBuildViews активирована, мы увидим сообщение об ошибке во время работы над проектом, как показано на рисунке 26-1.

**Рисунок 26-1:** Ошибка компилятора из-за проблем в представлении



Эта техника позволяет обнаруживать только ошибки компилятора. Она не видит логических ошибок и не может заменить полноценное тестирование, но в качестве предосторожности ее полезно применять перед развертыванием приложения. Мы не сможем развернуть приложение с ошибкой в представлении, так что мы восстановили его до прежнего состояния, как показано в листинге 26-3.

**Листинг 26-3:** Удаляем ошибку из представления List.cshtml

```
@model SportsStore.WebUI.Models.ProductsListViewModel

@{
    ViewBag.Title = "Products";
}

@foreach (var p in Model.Products)
{
    Html.RenderPartial("ProductSummary", p);
}



@Html.PagerLinks(Model.PagingInfo, x => Url.Action("List",
        new { page = x, category = Model.CurrentCategory }))


```

#### Подсказка

*Вы можете увидеть сообщение о том, что будет ошибкой будет использовать секцию, которая зарегистрирована как allowDefinition='MachineToApplication' вне уровня приложения. Это ошибка, которая возникает после развертывания приложения. Мы нашли только одно надежное решение для этой проблемы: очистить проект в режиме отладки, затем в режиме релиза, а затем построить приложение в режиме отладки.*

## Дезактивируем режим отладки

Одним из наиболее важных элементов в файле Web.config, на который вы должны обратить внимание при развертывании приложения, является compilation, как показано в листинге 26-4.

**Листинг 26-4:** Элемент compilation в Web.config

```
<system.web>
    <httpRuntime targetFramework="4.5" />
    <compilation debug="true" targetFramework="4.5" />
```

```
<authentication mode="Forms">
  <forms loginUrl("~/Account/Login" timeout="2880">
```

Когда атрибут `debug` содержит значение `true`, поведение компилятора и приложения будет ориентировано на процесс разработки. Например, компилятор будет выполнять следующие действия:

- Опускает некоторые процедуры оптимизации кода, чтобы он мог построчно выполнять код.
- Компилирует каждое представление по запросу, вместо компиляции всех представлений в один прием.
- Отключает лимит времени запроса, чтобы мы могли провести много времени в отладчике.
- Ограничивает способы кэширования контента браузерами.

Кроме того, если вы использовали связки для группировки одновременно и CSS, и JavaScript файлов, браузер получит инструкцию загружать каждый файл по отдельности, и он не получит минимизированные версии этих файлов. (Мы рассмотрели связки в главе 24).

Все эти функции полезны при разработке приложения, но они станут помехой при развертывании. Как вы можете догадаться, решение заключается в изменении значения атрибута `debug` на `false`:

```
<compilation debug="false" targetFramework="4.5" />
```

Чаще всего вам не придется выполнять это изменение перед развертыванием, поскольку изменить конфигурацию приложения можно в утилитах развертывания Visual Studio, или значение в файле `Web.config` будет переопределено конфигурацией сервера приложений IIS.

Тем не менее, если поведение приложения меняется, когда атрибут `debug` имеет значение `false`, то после отключения режима отладки необходимо запустить программу тестирования до того, как выполнять развертывание. Вы должны убедиться, что ваши представления визуализируются так, как вы и планировали, и все связки, использующие маркер `{version}`, указывают на файлы, которые существуют и доступны на сервере.

## Удаляем неиспользуемые строки подключения

Чаще всего в процессе развертывания ошибки возникают при настройке соединения с базой данных. Как мы знаем из своего опыта, большинство проблем возникают из-за информации о неиспользуемых базах данных в файле `web.config`.

Зная, что базы данных проблематичны, разработчики утилит развертывания пытаются автоматизировать процесс настройки баз данных, но они зациклены на конфигурации по умолчанию, которую добавляет в проект Visual Studio. Чтобы избежать этих проблем, можно отредактировать файл `Web.config`. В листинге 26-5 показаны два варианта настройки соединения для передачи данных в файле `Web.config` в проекте `sportsStore.WebUI`. Запись, в которой атрибут `name` содержит `DefaultConnection`, была создана Visual Studio; она помогает запустить новый проект, и мы не используем ее в приложении `SportsStore`. Другую запись - `EFDbContext` - создали и использовали мы, и она потребуется в развернутом проекте.

### Листинг 26-5: Строки подключения в файле `Web.config`

```
<configuration>
  <configSections>
    <section name="entityFramework"
      type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection,
      EntityFramework, Version=5.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
      requirePermission="false" />
```

```

</configSections>
<connectionStrings>
    <add name="DefaultConnection"
        providerName="System.Data.SqlClient"
        connectionString="Data Source=(LocalDb)\v11.0;Initial Catalog=aspnet-
SportsStore.WebUI-20121003232522;Integrated
Security=SSPI;AttachDBFilename=|DataDirectory|\aspnet-SportsStore.WebUI-
20121003232522.mdf" />
    <add name="EFDbContext"
        connectionString="Data Source=(localdb)\v11.0;Initial
Catalog=SportsStore;Integrated Security=True"
        providerName="System.Data.SqlClient"/>
</connectionStrings>

```

Вы можете просто закомментировать ненужные соединения, но мы предпочитаем вносить более определенные изменения, потому что мы провели много часов за отладкой проблем развертывания, возникших из-за информации о неиспользуемых подключениях. В листинге 26-6 показано, как мы отредактировали файл Web.config, оставив только запись в EFDbContext.

**Листинг 26-6:** Удаляем информацию о неиспользуемых подключениях

```

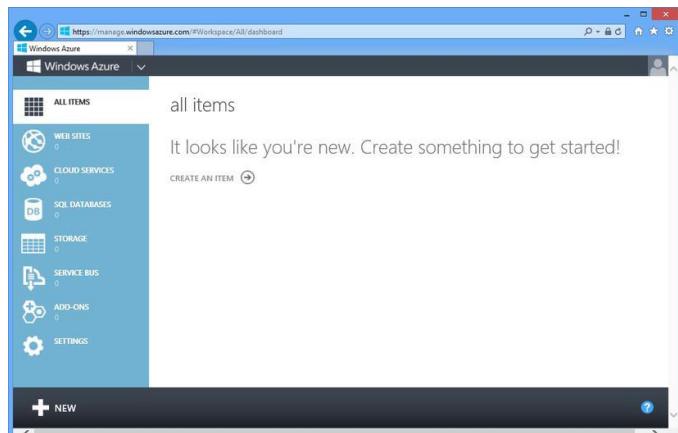
<configuration>
    <configSections>
        <section name="entityFramework"
            type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection,
EntityFramework, Version=5.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
            requirePermission="false" />
    </configSections>
    <connectionStrings>
        <add name="EFDbContext"
            connectionString="Data Source=(localdb)\v11.0;Initial
Catalog=SportsStore;Integrated Security=True"
            providerName="System.Data.SqlClient"/>
    </connectionStrings>

```

## Подготовка Windows Azure

После того, как мы подготовили и протестировали приложение, можно подготовить хостинг. Прежде чем использовать Azure, необходимо создать аккаунт на [www.windowsazure.com](http://www.windowsazure.com). На момент написания данной книги Microsoft предлагала бесплатные пробные аккаунты; некоторые пакеты MSDN также включают сервисы Azure. Когда вы создадите аккаунт, можно настраивать сервисы Azure, введя свои учетные данные на <http://manage.windowsazure.com>. Сначала вы увидите главную страницу, как показано на рисунке 26-2.

**Рисунок 26-2:** Портал Azure



### **Внимание!**

*На момент написания книги портал Azure работал только с Internet Explorer. Другие браузеры не отображали все всплывающие окна или приложение Silverlight, которое требуется для настройки базы данных.*

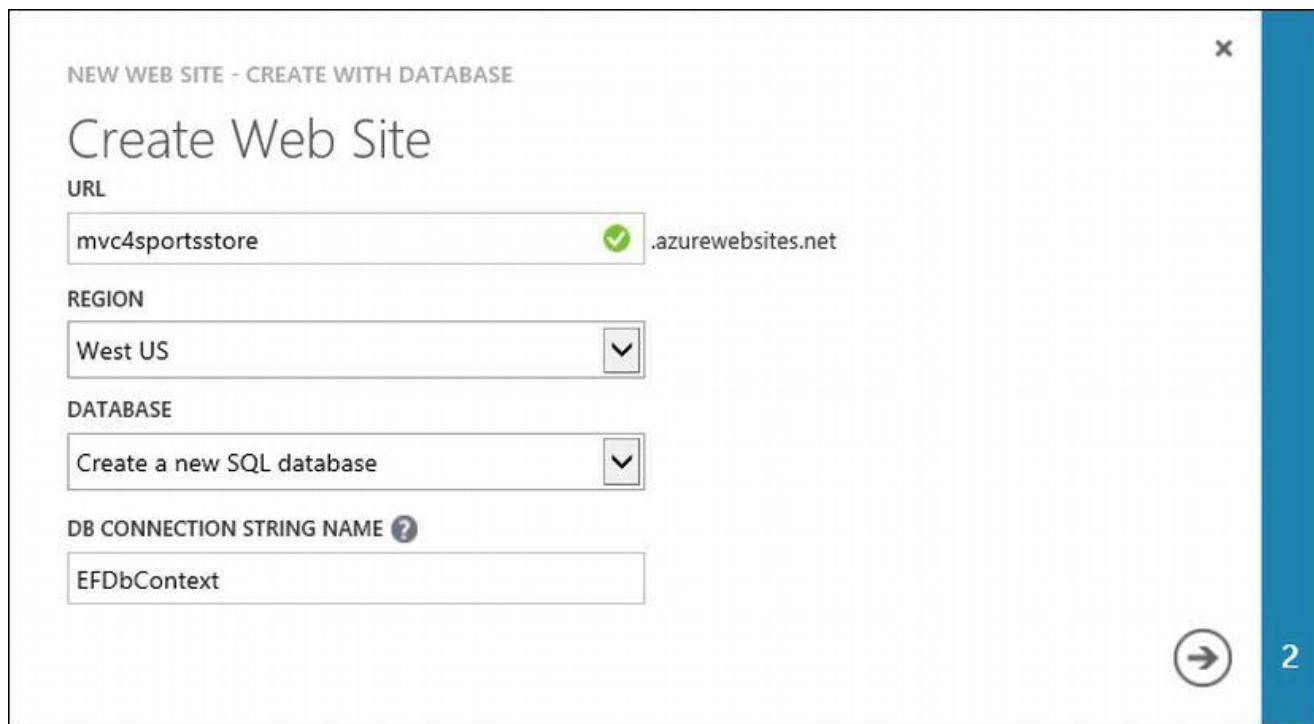
## **Создаем сайт и базу данных**

Начнем с создания нового сайта и сервиса базы данных. Нажмите на большой плюс в левом нижнем углу окна и выберите Compute - Web Site - Create With Database. Вы увидите форму, как показано на рисунке 26-3.

### **Подсказка**

*На момент написания книги был доступен предварительный просмотр для функции Web Site. Чтобы активировать данную функцию, нажмите на ссылку, которая появляется при нажатии на опцию Compute.*

**Рисунок 26-3:** Создаем новый сайт и базу данных



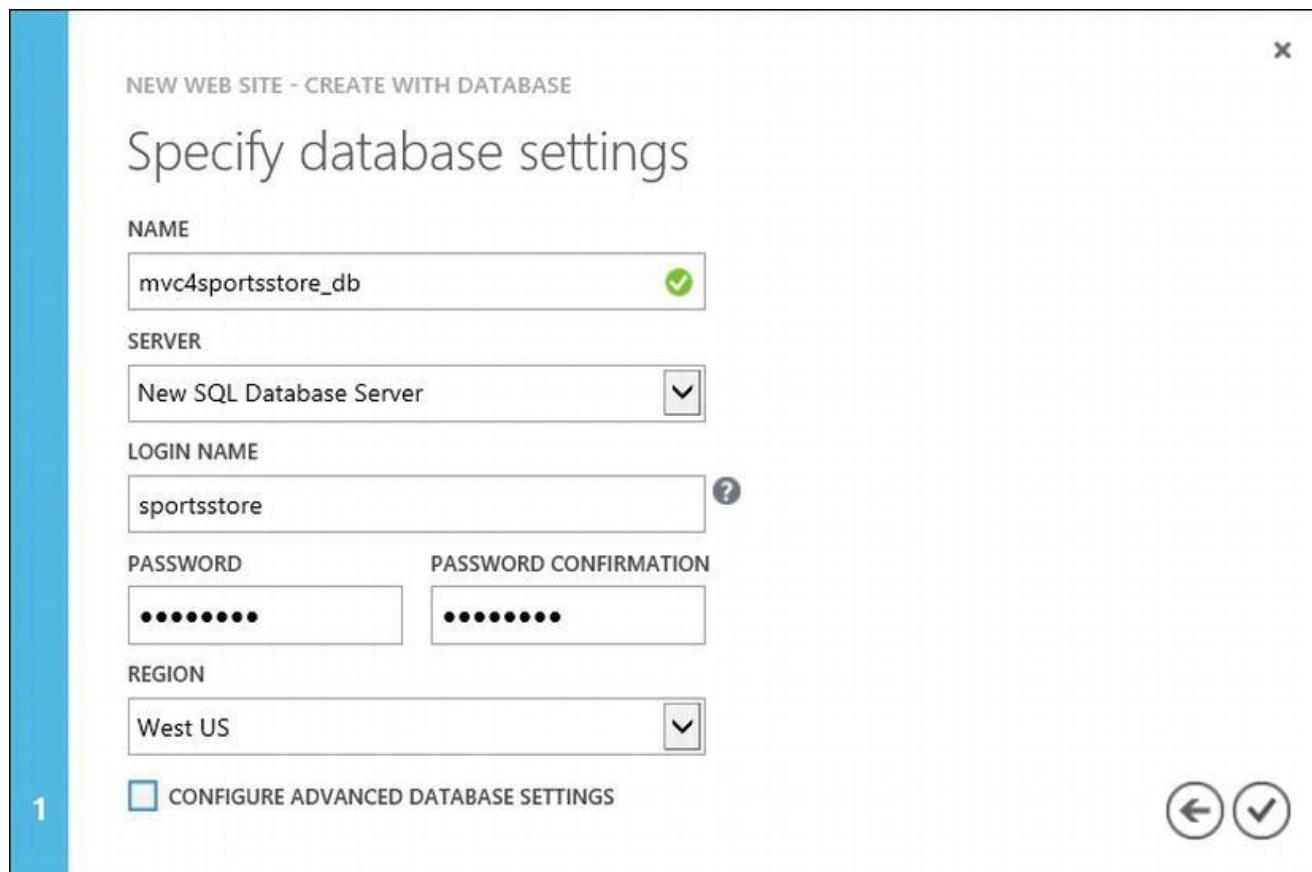
Нужно выбрать URL для нашего приложения. Для бесплатных и базовых сервисов Azure мы сможем выбрать только имена в домене `azurewebsites.net`. Мы назвали приложение `mvc4sportsstore`, но вам придется придумать что-то свое, так как для каждого сайта Azure требуется уникальное имя.

Выберите область, в которую вы хотите развернуть приложение, и отметьте опцию Create a new SQL database (в Azure можно использовать MySql, но наше приложение не настроено для работы с ней, поэтому мы выбираем базу данных SQL Server).

В поле DB Connection String Name введите `EFDbContext`. Приложение SportsStore использует это имя для установки соединения с базой данных, и, используя его в сервисе Azure, мы гарантируем,

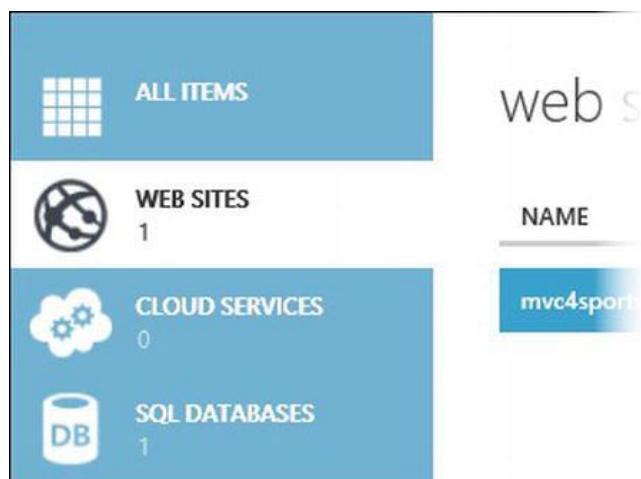
что код приложения будет работать после развертывания без модификаций. Когда вы заполните форму, нажмите кнопку со стрелкой, чтобы перейти к форме, показанной на рисунке 26-4.

**Рисунок 26-4:** Настраиваем базу данных



Выберите опцию New SQL Data Server в поле Server и введите имя пользователя и пароль. Мы указали имя sportsstore и создали пароль из букв обоих регистров и цифр, следуя рекомендациям из формы. Запишите ваши имя пользователя и пароль, потому что они понадобятся вам в следующем разделе. Нажмите на кнопку с галочкой, чтобы завершить процесс установки. Azure создаст новый сайт и базу данных, что может занять несколько минут. Когда установка будет завершена, вы будете перенаправлены на главную страницу, где увидите, что в категориях Web Sites и SQL Databases появилось по одному элементу, как показано на рисунке 26-5.

**Рисунок 26-5:** Результат создания сайта и базы данных

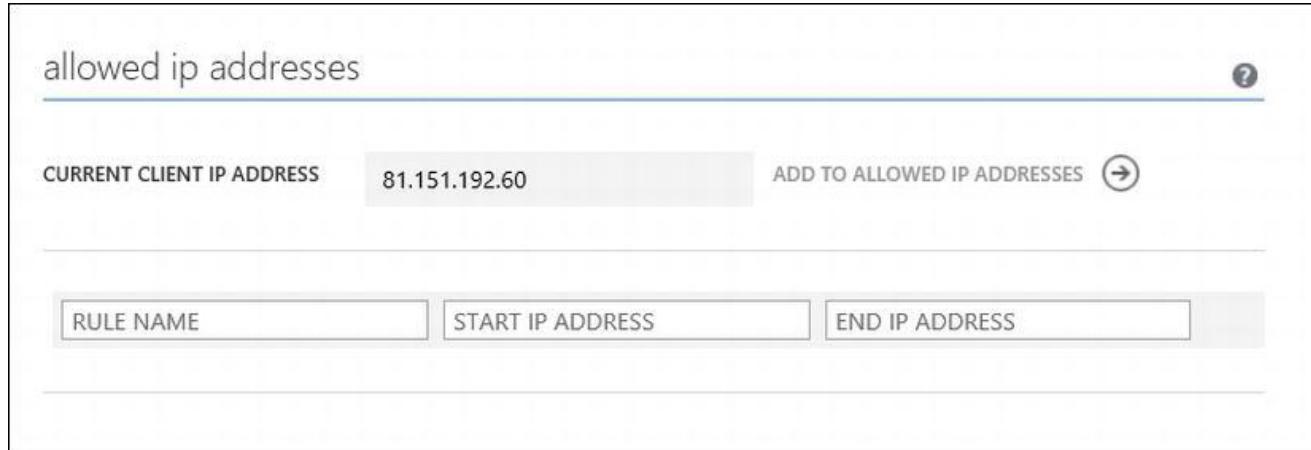


## Подготовка базы данных к удаленному администрированию

Следующим шагом будет настройка базы данных Azure так, чтобы она следовала той же схеме и содержала те же данные, которые мы использованы в главе 7. Нажмите на ссылку SQL Databases на главной странице Azure и затем выберите запись, которая появляется в таблице SQL Databases (если вы приняли значения по умолчанию, база данных будет называться mvc4sportsstore\_db).

Вы увидите подробную информацию о базе данных и ее производительности (о которой ничего не известно, потому что в базе еще нет контента и к ней еще не поступало никаких запросов). Нажмите на ссылку Manage allowed IP addresses в разделе Quick Glance, и вы увидите форму, показанную на рисунке 26-6.

**Рисунок 26-6:** Активируем доступ к конфигурации в фаерволе



Azure ограничивает доступ к базам данных, чтобы они были доступны только для других сервисов Azure. Нам нужно разрешить доступ нашему рабочему компьютеру, для чего мы нажимаем Add to Allowed IP Addresses и затем – кнопку Save, которая появляется в нижней части окна браузера.

### Подсказка

*Вам нужно будет добавить IP-адреса всех клиентских компьютеров, с которых вы хотите администрировать базу данных Azure.*

Нажмите на ссылку Dashboard в верхней части страницы, а затем - на ссылку, которая отображается под Manage URL. Откроется новое окно браузера и загрузится утилита администрирования базы данных Silverlight.

### Подсказка

*Если у вас еще нет Silverlight, нужно будет его установить. Браузер предложит вам провести инсталляцию и проведет через процесс установки автоматически.*

Оставьте поле Database пустым и введите учетные данные, которые вы использовали в предыдущем разделе, чтобы перейти к администрированию базы данных. Если вы увидите сообщение о том, что при подключении к серверу произошла ошибка, подождите несколько минут и повторите попытку. Так происходит потому, что правило фаервола, которое разрешает вашему компьютеру доступ к базе данных, может распространяться в инфраструктуре Azure в течение нескольких минут.

## Создаем схему

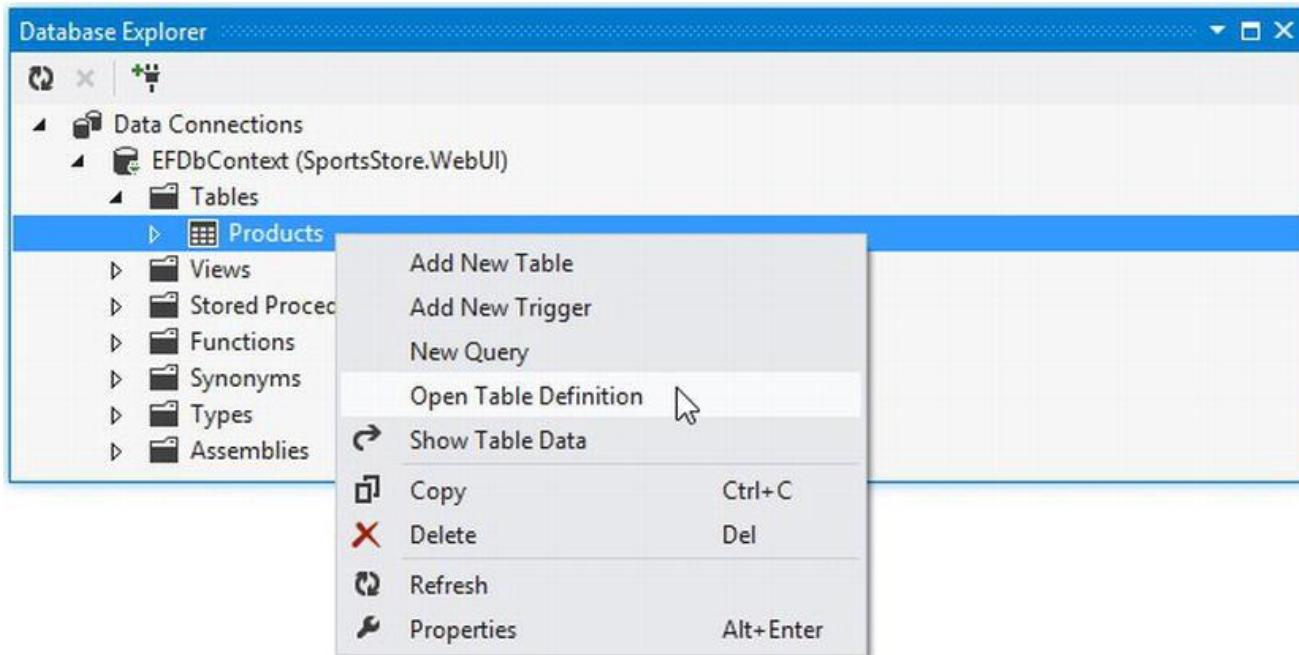
Нашим следующим шагом будет создание схемы для базы данных. Нажмите кнопку Administration и вы увидите в основной части окна браузера элемент, представляющий базу данных, которую мы создали ранее (если вы приняли имя по умолчанию, она будет называться `mvc4sportsstore.db`).

В нижней части элемента базы данных есть несколько небольших кнопок. Найдите и нажмите на кнопку Query, после чего появится пустая текстовая область. Здесь мы введем команду SQL, которая создаст нужную нам таблицу базы данных.

### Получаем команды для создания схемы

Мы можем получить необходимые команды SQL из Visual Studio. Откройте окно Database Explorer и раскрывайте его элементы, пока не найдете запись для таблицы Products. Кликните по таблице правой кнопкой мыши и выберите Open Table Definition, как показано на рисунке 26-7.

Рисунок 26-7: Получаем определение таблицы в окне Data Explorer



Откроется редактор для схемы таблицы. В таблице T-SQL вы увидите код SQL, показанный в листинге 26-7.

Листинг 26-7: Операторы для создания таблицы Products

```
CREATE TABLE [dbo].[Products] (
    [ProductID] INT IDENTITY (1, 1) NOT NULL,
    [Name] NVARCHAR (100) NOT NULL,
    [Description] NVARCHAR (500) NOT NULL,
    [Category] NVARCHAR (50) NOT NULL,
    [Price] DECIMAL (16, 2) NOT NULL,
    [ImageData] VARBINARY (MAX) NULL,
    [ImageMimeType] VARCHAR (50) NULL,
    PRIMARY KEY CLUSTERED ([ProductID] ASC)
);
```

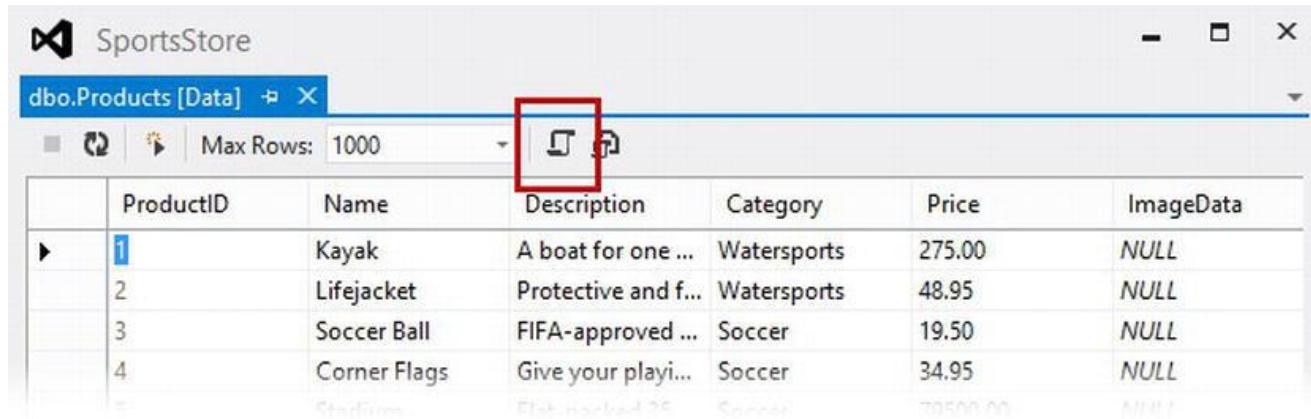
Скопируйте SQL из Visual Studio, вставьте его в текстовую область в браузере и нажмите кнопку Run в верхней части окна браузера. Через пару секунд вы увидите сообщение Command(s) completed

successfully, что свидетельствует о том, что наша база данных Azure содержит базу Product с той же схемой, которую мы определили в приложении SportsStore.

## Добавляем данные в таблицу

Теперь, когда мы создали таблицу, мы можем заполнить ее данными о товарах, которые мы использовали в главе 7. Вернитесь к записи Products в окне Database Explorer, кликните по ней правой кнопкой мыши и выберите Show Table Data из контекстного меню. В верхней части открывшегося окна вы найдете кнопку Script, как показано на рисунке 26-8.

Рисунок 26-8: Кнопка Script в окне с информацией о таблице



ProductID	Name	Description	Category	Price	ImageData
1	Kayak	A boat for one ...	Watersports	275.00	NULL
2	Lifejacket	Protective and f...	Watersports	48.95	NULL
3	Soccer Ball	FIFA-approved ...	Soccer	19.50	NULL
4	Corner Flags	Give your playi...	Soccer	34.95	NULL
	Stadium	Flat-packed 35,000-seat stadium	Soccer	79500.00	NULL
	Thinking Cap	Improve your brain efficiency by 75%	Chess	16.00	image/jpeg
	Unsteady Chair	Secretly give your opponent a disadvantage	Chess	29.95	NULL
	Human Chess Board	A fun game for the family	Chess	40.00	NULL

Откроется новое окно, содержащее другой оператор SQL, показанный в листинге 26-8.

Листинг 26-8: Оператор SQL для добавления данных в таблицу Products

```
SET IDENTITY_INSERT [dbo].[Products] ON
INSERT INTO [dbo].[Products] ([ProductID], [Name], [Description], [Category], [Price],
[ImageMimeType]) VALUES (1, N'Kayak', N'A boat for one person', N'Watersports',
CAST(275.00 AS
Decimal(16, 2)), NULL)
INSERT INTO [dbo].[Products] ([ProductID], [Name], [Description], [Category], [Price],
[ImageMimeType]) VALUES (2, N'Lifejacket', N'Protective and fashionable',
N'Watersports',
CAST(48.95 AS Decimal(16, 2)), NULL)
INSERT INTO [dbo].[Products] ([ProductID], [Name], [Description], [Category], [Price],
[ImageMimeType]) VALUES (3, N'Soccer Ball', N'FIFA-approved size and weight',
N'Soccer',
CAST(19.50 AS Decimal(16, 2)), NULL)
INSERT INTO [dbo].[Products] ([ProductID], [Name], [Description], [Category], [Price],
[ImageMimeType]) VALUES (4, N'Corner Flags', N'Give your playing field a professional
touch',
N'Soccer', CAST(34.95 AS Decimal(16, 2)), NULL)
INSERT INTO [dbo].[Products] ([ProductID], [Name], [Description], [Category], [Price],
[ImageMimeType]) VALUES (5, N'Stadium', N'Flat-packed 35,000-seat stadium', N'Soccer',
CAST(79500.00 AS Decimal(16, 2)), NULL)
INSERT INTO [dbo].[Products] ([ProductID], [Name], [Description], [Category], [Price],
[ImageMimeType]) VALUES (6, N'Thinking Cap', N'Improve your brain efficiency by 75%',
N'Chess',
CAST(16.00 AS Decimal(16, 2)), N'image/jpeg')
INSERT INTO [dbo].[Products] ([ProductID], [Name], [Description], [Category], [Price],
[ImageMimeType]) VALUES (7, N'Unsteady Chair', N'Secretly give your opponent a
disadvantage',
N'Chess', CAST(29.95 AS Decimal(16, 2)), NULL)
INSERT INTO [dbo].[Products] ([ProductID], [Name], [Description], [Category], [Price],
[ImageMimeType]) VALUES (8, N'Human Chess Board', N'A fun game for the family',
N'Chess',
```

```

CAST(75.00 AS Decimal(16, 2)), NULL)
INSERT INTO [dbo].[Products] ([ProductID], [Name], [Description], [Category], [Price],
[ImageMimeType]) VALUES (10, N'Bling-Bling King', N'Gold-plated, diamond-studded King',
N'Chess', CAST(1200.00 AS Decimal(16, 2)), NULL)
SET IDENTITY_INSERT [dbo].[Products] OFF

```

Очистите текстовую область в окне Azure и вставьте SQL из листинга. Нажмите кнопку Run. Скрипт будет выполнен и добавит данные в таблицу.

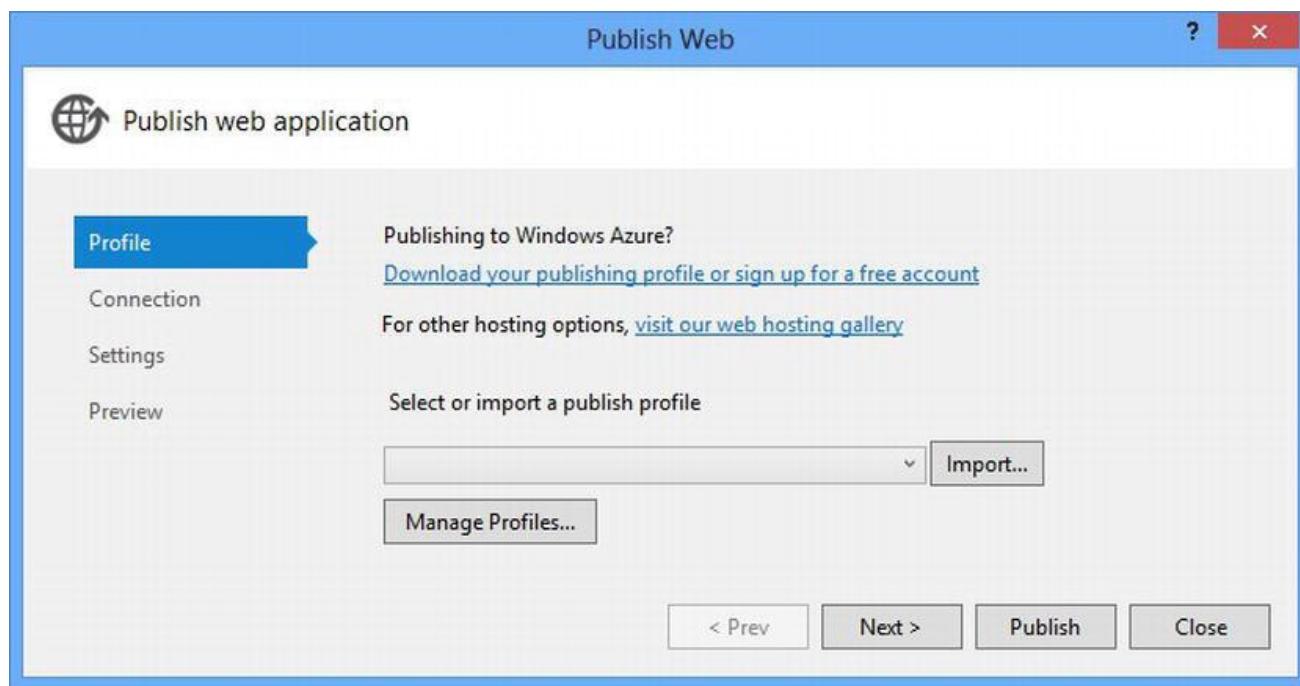
## Публикация приложения

Теперь, когда установка завершена, развертывание приложения будет относительно простым. Вернитесь на главный портал Azure и нажмите кнопку Web Sites. Нажмите на сайт mvc4sportsstore, чтобы открыть панель инструментов, а затем нажмите на ссылку Download publish profile в разделе Quick Glance. Сохраните этот файл там, где сможете его найти.

Для нашего сервиса Azure этот файл называется mvc4sportsstore.azurewebsites.net.PublishSettings. Мы сохранили его на рабочем столе. Этот файл содержит информацию, которая понадобится Visual Studio для публикации приложения в инфраструктуре Azure.

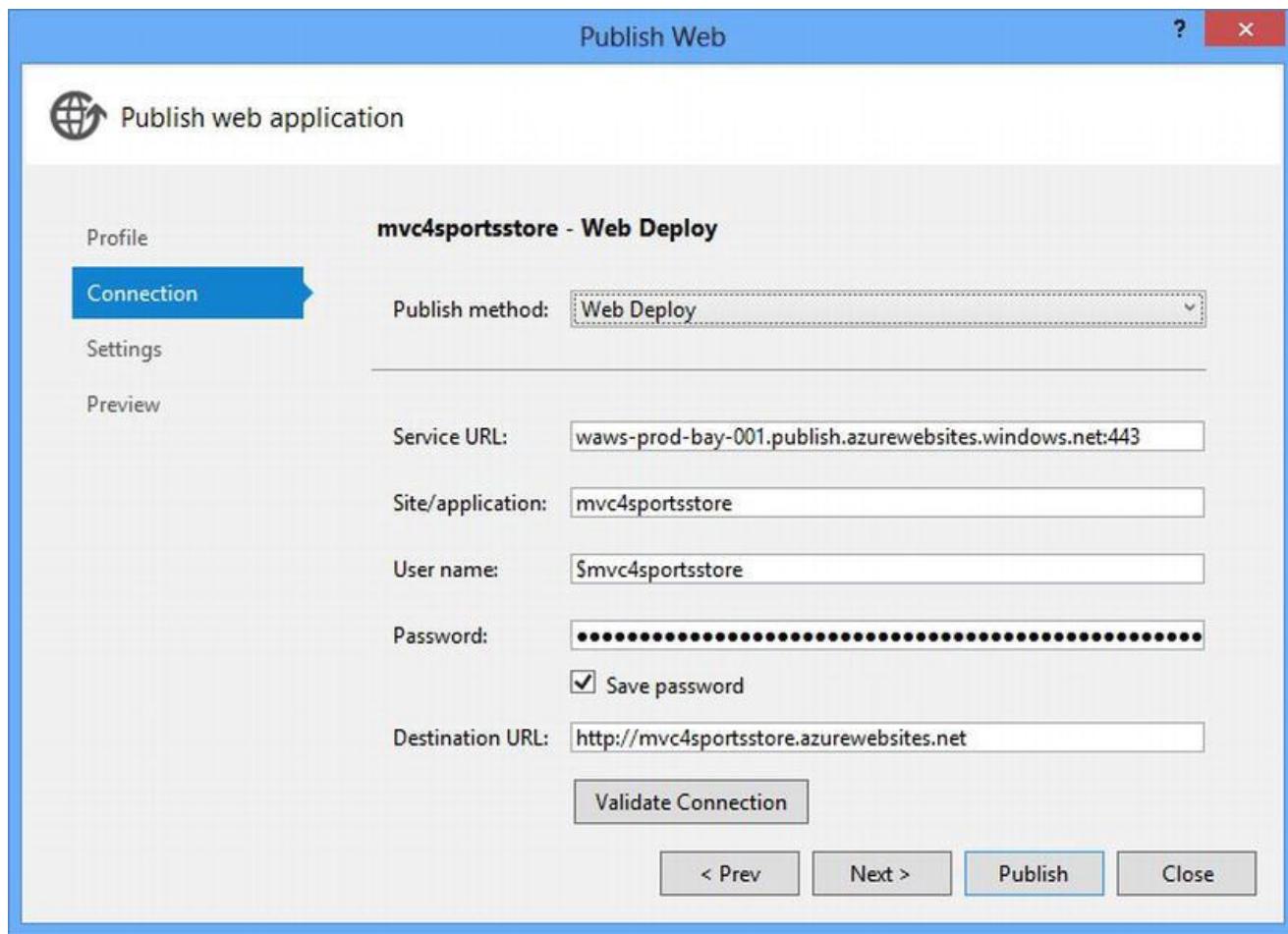
Кликните правой кнопкой мыши по проекту SportsStore.WebUI в окне Solution Explorer и выберите Publish из контекстного меню. Вы увидите диалоговое окно Publish Web, как показано на рисунке 26-9.

**Рисунок 26-9:** Диалоговое окно Publish Web



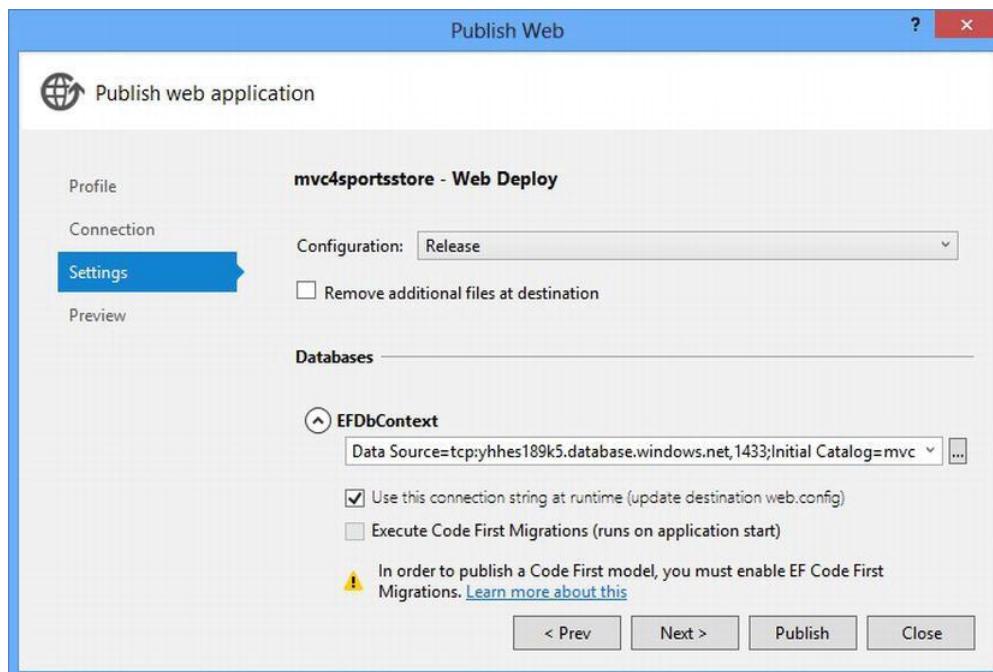
Нажмите кнопку Import и выберите файл, который вы скачали с портала Azure. Visual Studio обработает этот файл и отобразит информацию о конфигурации вашего сервиса Azure, как показано на рисунке 26-10. Вы увидите то имя, которое выбрали для своего сайта.

**Рисунок 26-10:** Подробная информация о сервисе Azure, в который будет развернуто приложение



Нет необходимости изменять отображенные значения. Нажмите кнопку Next, чтобы перейти к следующему этапу процесса развертывания, который показан на рисунке 26-11.

**Рисунок 26-11:** Настройки для развернутого приложения

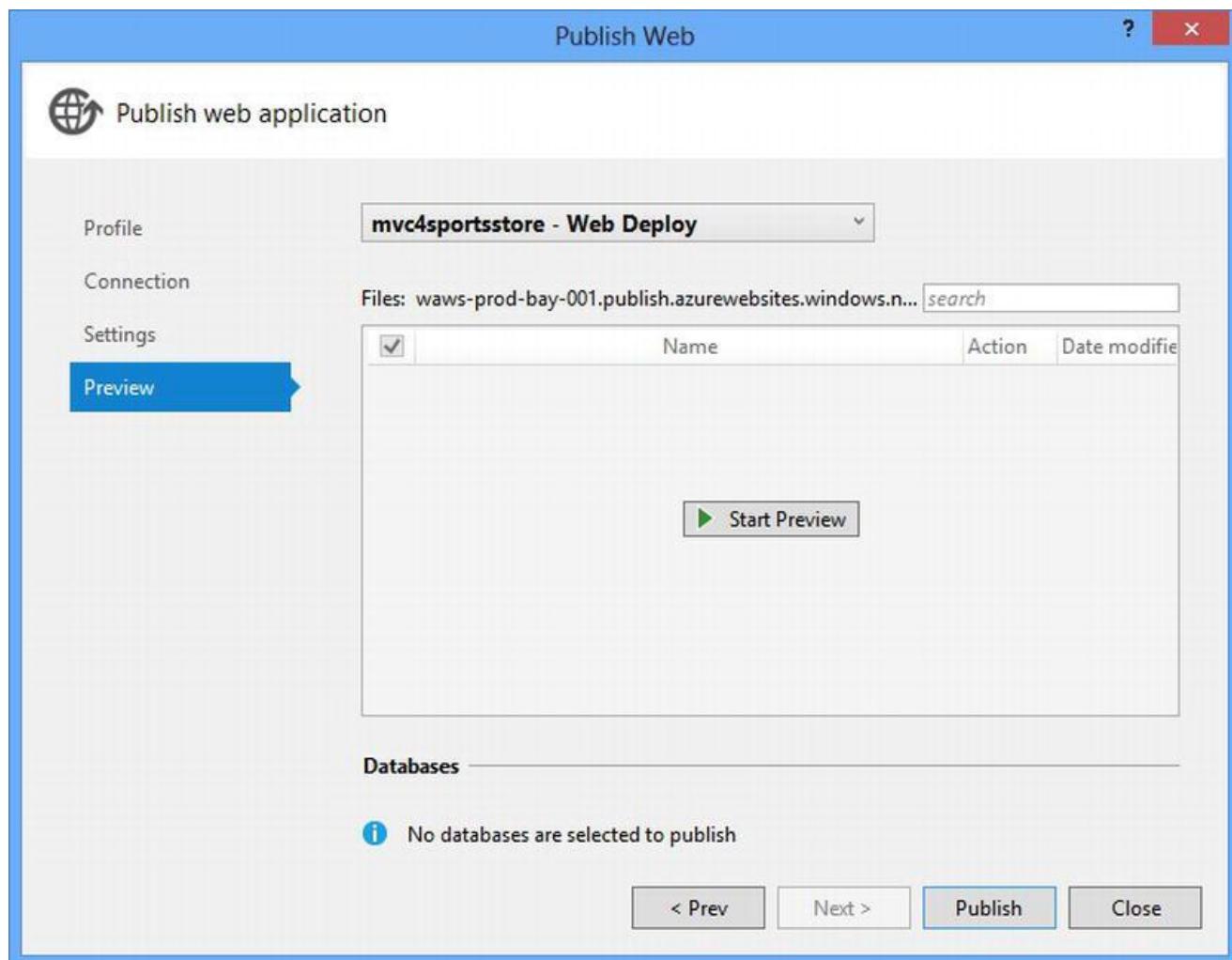


Можно выбрать конфигурацию, которая будет использоваться при развертывании. Обычно это будет Release, но вы можете выбрать Debug, если собираетесь тестировать приложение в инфраструктуре Azure и хотите увидеть настройки отладки для компилятора и связок приложения.

Другая часть этого процесса – это конфигурация подключений к базе данных. Visual Studio дает возможность установить соответствие между подключением к базе данных, определенным в проекте, и базой данных, которая связана с сайтом Azure. Мы гарантировали, что файл Web.config содержит только один вариант подключения, и, так как мы создали только одну базу данных Azure, нам подойдет соответствие по умолчанию. Если в вашем приложении несколько баз данных, вы должны позаботиться о том, с каждым подключением в приложении была связана правильная база данных Azure.

Нажмите кнопку Next, чтобы предварительно просмотреть эффект развертывания, как показано на рисунке 26-12. Когда вы нажмете кнопку Start Preview, Visual Studio симулирует процесс развертывания, но на самом деле не отправит файлы на сервер. Если вы обновляете приложение, которое уже развернуто, это может быть полезным, чтобы убедиться, что вы только заменили только те файлы, которые планировали.

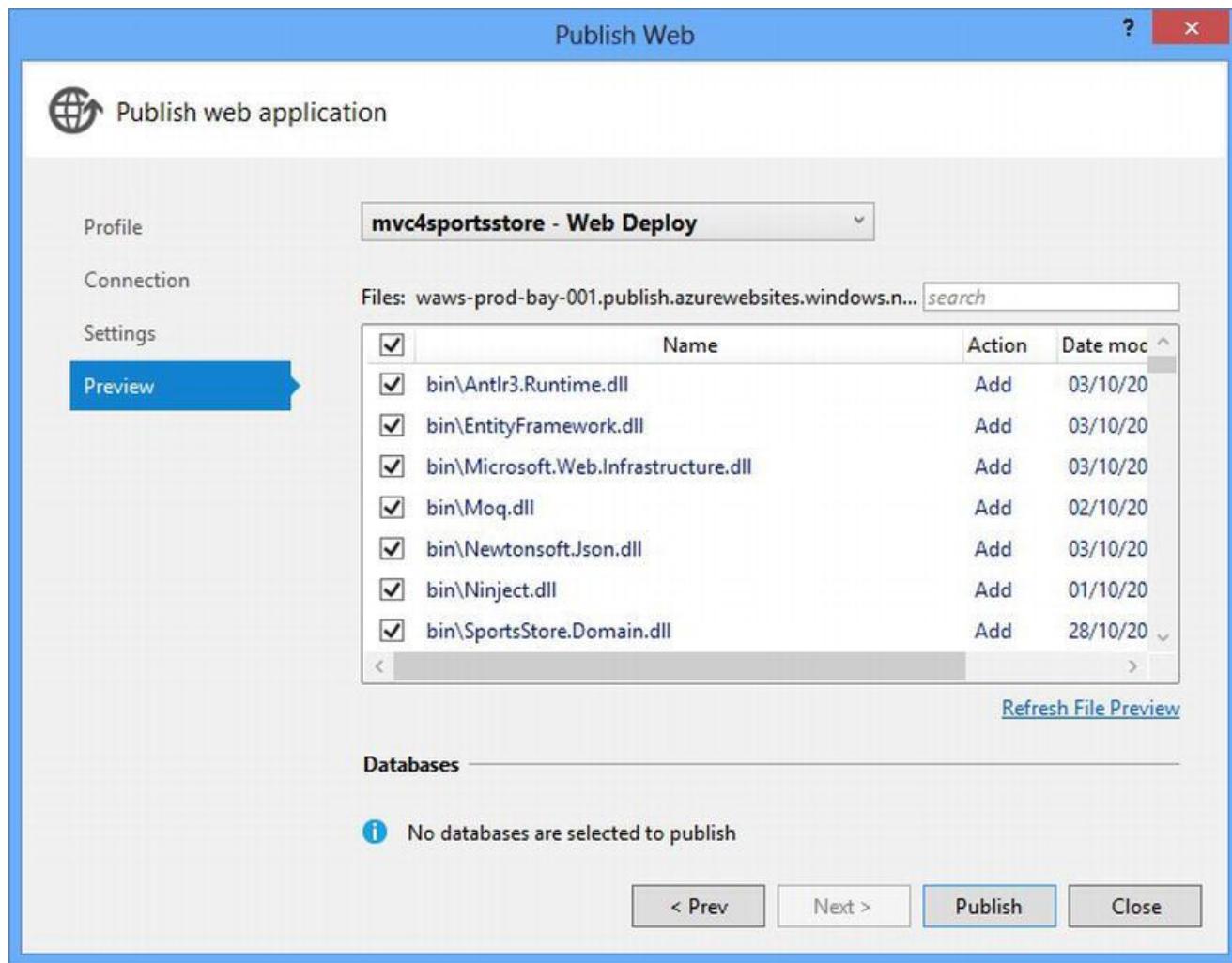
**Рисунок 26-12:** Раздел Preview диалогового окна Publish Web



Мы развертываем приложение в первый раз, поэтому в окне предварительного просмотра появятся все файлы проекта, как показано на рисунке 26-13. Обратите внимание, что напротив каждого файла есть чекбокс. Вы можете отменить развертывание отдельных файлов, хотя это нужно делать очень

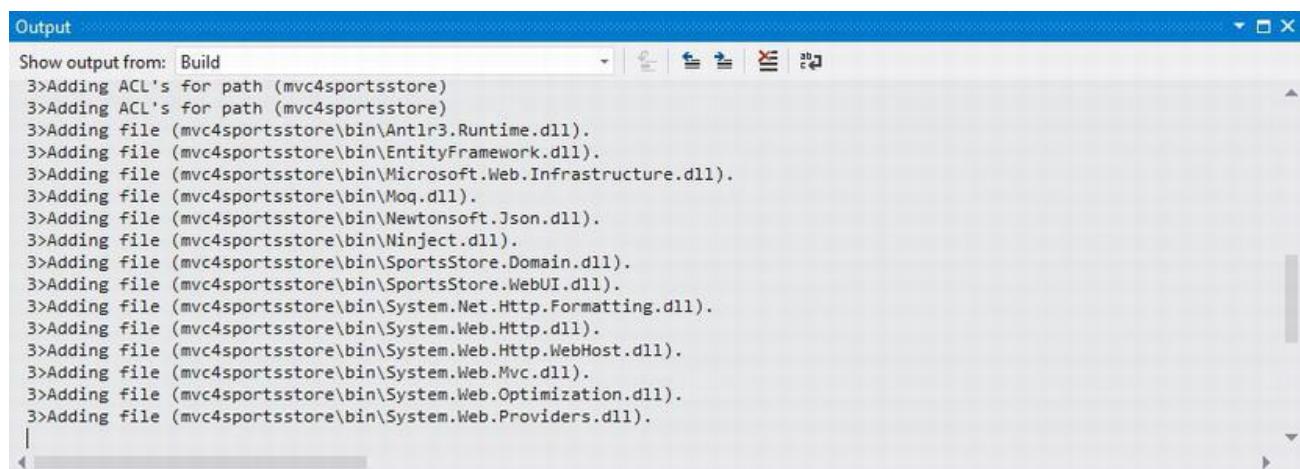
осторожно. Мы довольно консервативны в этом отношении, и предпочитаем развернуть файлы, которые нам не нужны, чем забыть один нужный файл.

**Рисунок 26-13:** Просматриваем изменения развертывания



Нажмите кнопку Publish, чтобы развернуть приложение на платформе Azure. Диалоговое окно Publish Web закроется, и вы сможете просмотреть информацию о прогрессе развертывания окне Output в Visual Studio, как показано на рисунке 26-14.

**Рисунок 26-14:** Развертываем приложение на платформе Azure

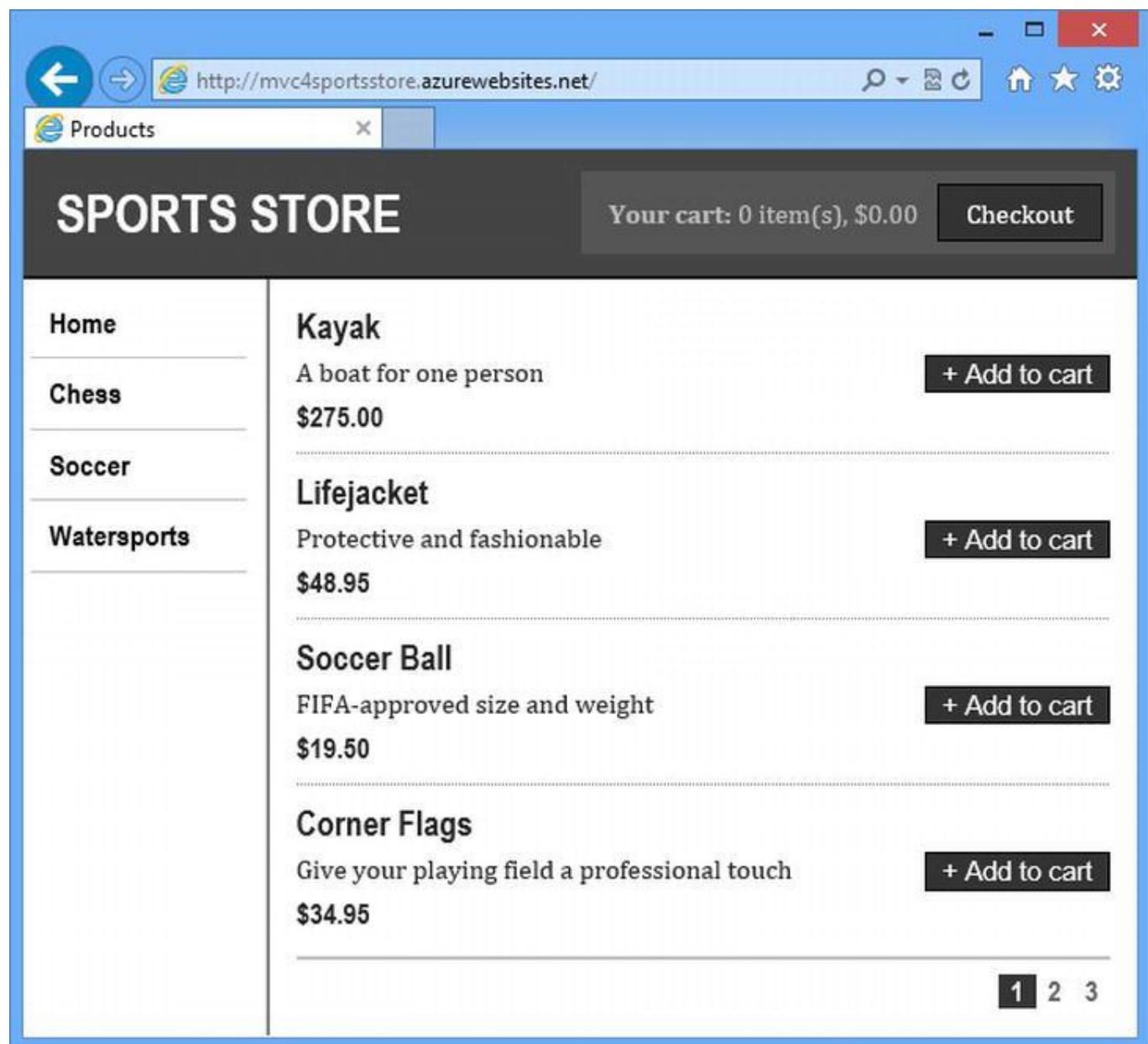


## Подсказка

Вы можете увидеть сообщение о том, что будет ошибкой будет использовать секцию, которая зарегистрирована как `allowDefinition='MachineToApplication'` вне уровня приложения. Это баг, который возникает после развертывания приложения. Мы нашли только одно надежное решение для этой проблемы: очистить проект в режиме отладки, затем в режиме релиза, а затем построить приложение в режиме отладки. Если вы перестроите проект, то сможете повторить развертывание. Диалоговое окно `Publish Web` запомнит параметры развертывания и перейдет прямо к предварительному просмотру.

Развертывание приложения может занять несколько минут, после чего процесс будет завершен. Visual Studio откроет окно браузера и перейдет по ссылке для вашего сайта Azure. Для нас это `http://mvc4sportsstore.azurewebsites.net`, как показано на рисунке 26-15.

Рисунок 26-15: Приложение SportsStore, работающее на платформе Windows Azure



# Резюме

В этой главе мы показали вам, как подготовить приложение к развертыванию, а также как создать простой сервис Windows Azure и развернуть приложение на нем. Есть много различных способов развертывания приложений и много целевых платформ, но процесс, который мы показали вам в этой главе, иллюстрирует ключевые моменты развертывания, характерные не только для Azure.

Это все, что мы можем рассказать вам о MVC Framework. Мы начали с создания простого приложения, затем объяснили принципы работы различных компонентов платформы и показали, как их можно сконфигурировать, изменить или заменить полностью. Мы желаем вам всяческих успехов с вашими проектами MVC Framework и надеемся, что вы получили такое же удовольствие от чтения этой книги, какое мы получили от ее написания.