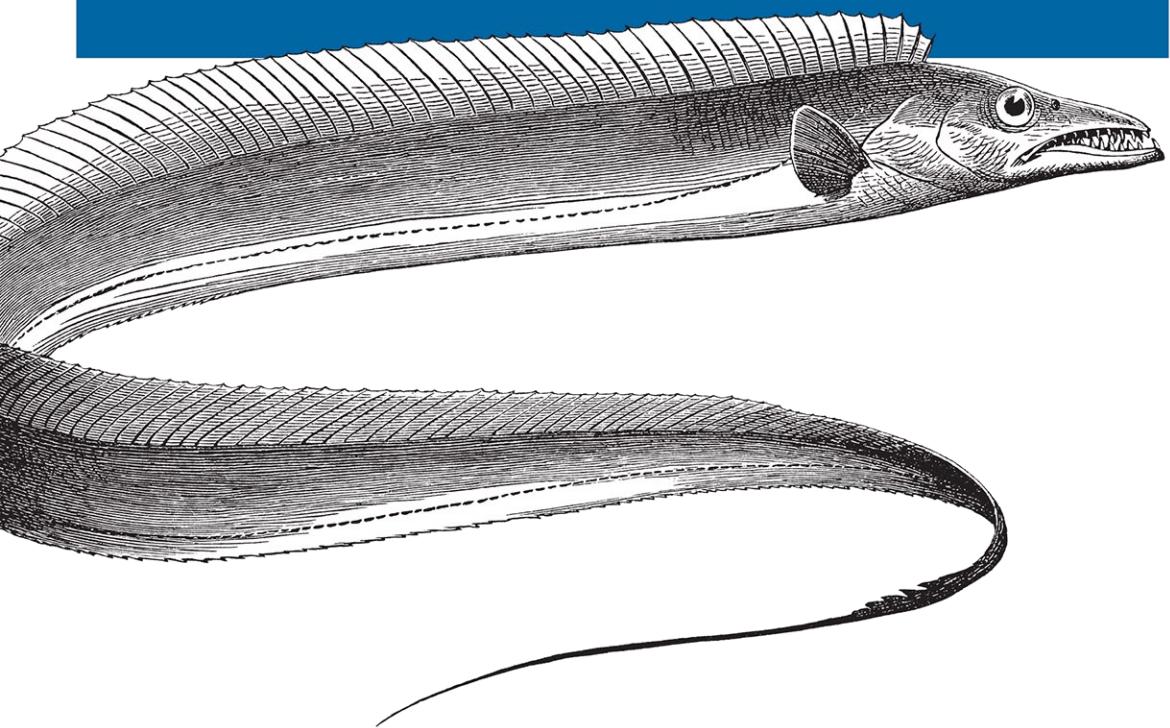


ASP.NET MVC 4



ASP.NET MVC 4:
разработка реальных
веб-приложений с
помощью ASP.NET MVC

Programming ASP.NET MVC 4

Jess Chadwick, Todd Snyder, and Hrusikesh Panda

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

ASP.NET MVC 4: разработка реальных веб-приложений с помощью ASP.NET MVC

Джесс Чедвик, Todd Снайдер, Хришикеш Панда



Издательский дом “Вильямс”
Москва • Санкт-Петербург • Киев 2013

ББК 32.973.26-018.2.75

Ч-35

УДК 681.3.07

Издательский дом “Вильямс”

Зав. редакцией С.Н. Тригуб

Перевод с английского Ю.Н. Артеменко

Под редакцией Ю.Н. Артеменко

По общим вопросам обращайтесь в Издательский дом “Вильямс” по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Чедвик, Джесс, Снайдер, Todd, Панда, Хришикеш.

Ч-35 ASP.NET MVC 4: разработка реальных веб-приложений с помощью ASP.NET MVC. :
Пер. с англ. — М. : ООО “И.Д. Вильямс”, 2013. — 432 с. : ил. — Парал. тит. англ.
ISBN 978-5-8459-1841-3 (рус.)

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками
соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой
бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические,
включая фотокопирование и запись на магнитный носитель, если на это нет письменного
разрешения издательства O'Reilly Media, Inc.

Authorized Russian translation of the English edition of *Programming ASP.NET MVC 4: Developing Real-World Web Applications with ASP.NET MVC* (ISBN 9781449320317) © 2012 Jess Chadwick, Todd Snyder, Hrusikesh Panda.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls
all rights to publish and sell the same.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any
means, electronic or mechanical, including photocopying, recording, or by any information storage or
retrieval system, without the prior written permission of the copyright owner and the Publisher.

Научно-популярное издание

Джесс Чедвик, Todd Снайдер, Хришикеш Панда

ASP.NET MVC 4: разработка реальных веб-приложений с помощью ASP.NET MVC

Верстка Т.Н. Артеменко
Художественный редактор В.Г. Павлютин

Подписано в печать 20.03.2013. Формат 70×100/16.

Гарнитура Times. Печать офсетная.

Усл. печ. л. 34,83. Уч.-изд. л. 25,2.

Тираж 1500 экз. Заказ № 0000.

Первая Академическая типография “Наука”
199034, Санкт-Петербург, 9-я линия, 12/28

ООО “И. Д. Вильямс”, 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-1841-3 (рус.)

ISBN 978-1-4493-2031-7 (англ.)

© 2013, Издательский дом “Вильямс”

© 2012, Jess Chadwick, Todd Snyder, Hrusikesh Panda

Оглавление

Часть I. Начало работы	19
Глава 1. Основы ASP.NET MVC	20
Глава 2. ASP.NET MVC для разработчиков Web Forms	56
Глава 3. Работа с данными	67
Глава 4. Разработка на стороне клиента	78
Часть II. Переход на следующий уровень	93
Глава 5. Архитектура веб-приложений	94
Глава 6. Улучшение сайта с помощью AJAX	117
Глава 7. ASP.NET Web API	143
Глава 8. Расширенная работа с данными	156
Глава 9. Безопасность	177
Глава 10. Разработка веб-приложений для мобильных устройств	200
Часть III. Выход за стандартные рамки	221
Глава 11. Параллельные, асинхронные и операции над данными в реальном времени	222
Глава 12. Кеширование	235
Глава 13. Технологии оптимизации клиентской стороны	257
Глава 14. Расширенная маршрутизация	279
Глава 15. Многократно используемые компоненты пользовательского интерфейса	298
Часть IV. Контроль качества	311
Глава 16. Регистрация в журнале	312
Глава 17. Автоматизированное тестирование	322
Глава 18. Автоматизация построения	354
Часть V. Выход в свет	367
Глава 19. Разворачивание	368
Часть VI. Приложения	383
Приложение А. Интеграция ASP.NET MVC и Web Forms	384
Приложение Б. Использование NuGet в качестве платформы	391
Приложение В. Рекомендуемые приемы	409
Приложение Г. Перекрестные ссылки: целевые темы, функциональные возможности и сценарии	421
Предметный указатель	424

Содержание

Об авторах	15
Предисловие	16
Предполагаемая читательская аудитория	16
Что требуется для работы с этой книгой	16
Соглашения, используемые в этой книге	17
Использование примеров кода	17
От издательства	17
Часть I. Начало работы	19
Глава 1. Основы ASP.NET MVC	20
Платформа веб-разработки от Microsoft	20
Active Server Pages (ASP)	20
ASP.NET Web Forms	21
ASP.NET MVC	21
Архитектура “модель-представление-контроллер”	21
Модель	22
Представление	22
Контроллер	23
Нововведения, появившиеся в версии ASP.NET MVC 4	23
Введение в ЕВиУ	24
Установка ASP.NET MVC	25
Создание приложения ASP.NET MVC	26
Шаблоны проекта	26
Соглашение по конфигурации	29
Запуск приложения	30
Маршрутизация	30
Конфигурирование маршрутов	31
Контроллеры	33
Действия контроллеров	33
Результаты действий	34
Параметры действий	35
Фильтры действий	37
Представления	38
Определение местоположения представлений	38
Механизм Razor	39
Различие кода и разметки	40
Компоновки	41
Частичные представления	42
Отображение данных	44
Вспомогательные методы HTML и URL	46
Модели	46

Собираем все вместе	47
Маршрут	47
Контроллер	47
Представление	50
Аутентификация	53
Класс AccountController	54
Резюме	55
Глава 2. ASP.NET MVC для разработчиков Web Forms	56
Все это просто ASP.NET	56
Инструменты, языки и API-интерфейсы	56
Обработчики и модули HTTP	57
Управление состоянием	57
Развертывание и исполняющая среда	58
Больше различий, чем сходства	58
Разделение логики приложения и логики представления	59
URL и маршрутизация	59
Управление состоянием	60
Визуализация HTML-разметки	61
Реализация представлений ASP.NET MVC с использованием синтаксиса Web Forms	64
Несколько предостережений	65
Резюме	66
Глава 3. Работа с данными	67
Построение формы	67
Обработка отправок формы	69
Сохранение данных в базе	69
Entity Framework Code First: соглашение по конфигурации	70
Создание уровня доступа к данным с помощью Entity Framework Code First	70
Проверка достоверности данных	71
Указание бизнес-правил с помощью аннотаций данных	72
Отображение сообщений об ошибках проверки достоверности	75
Резюме	77
Глава 4. Разработка на стороне клиента	78
Работа с JavaScript	78
Селекторы	80
Реагирование на события	83
Манипулирование DOM	85
AJAX	86
Проверка достоверности на стороне клиента	88
Резюме	91
Часть II. Переход на следующий уровень	93
Глава 5. Архитектура веб-приложений	94
Шаблон “модель-представление-контроллер”	94
Разделение ответственности	94
MVC и веб-платформы	95

Разработка архитектуры веб-приложения	97
Логическое проектирование	97
Логическое проектирование веб-приложения ASP.NET MVC	97
Полезные советы по логическому проектированию	99
Физическое проектирование	100
Пространства имен проекта и имена сборок	100
Варианты развертывания	101
Полезные советы по физическому проектированию	101
Принципы проектирования	103
SOLID	103
Инверсия управления	108
Принцип Don't Repeat Yourself	115
Резюме	116
Глава 6. Улучшение сайта с помощью AJAX	117
Частичная визуализация	117
Визуализация частичных представлений	118
Визуализация с помощью JavaScript	123
Визуализация данных JSON	123
Запрашивание данных JSON	125
Шаблоны клиентской стороны	125
Повторное использование логики в запросах AJAX и не AJAX	128
Реагирование на запросы AJAX	129
Реагирование на запросы JSON	130
Применение одной и той же логики во множестве действий контроллера	131
Отправка данных на сервер	132
Отправка сложных объектов JSON	133
Выбор связывателя модели	135
Эффективная отправка и получение данных JSON	137
Междоменный AJAX	137
JSONP	137
Включение разделения ресурсов между источниками	141
Резюме	142
Глава 7. ASP.NET Web API	143
Построение службы данных	143
Регистрация маршрутов Web API	145
Соблюдение соглашения по конфигурации	145
Переопределение соглашений	146
Подключение Web API	147
Разбиение на страницы и запрашивание данных	149
Обработка исключений	150
Форматеры носителей	152
Резюме	155
Глава 8. Расширенная работа с данными	156
Шаблоны доступа к данным	156
Традиционные объекты CLR	156
Использование шаблона Repository	157

Объектно-реляционные отображатели	159
Обзор Entity Framework	160
Выбор подхода доступа к данным	161
Параллелизм базы данных	162
Построение уровня доступа к данным	164
Использование подхода Entity Framework Code First	164
Бизнес-модель предметной области EBuy	166
Работа с контекстом данных	169
Сортировка, фильтрация и разбиение данных на страницы	170
Резюме	176
Глава 9. Безопасность	177
Построение защищенных веб-приложений	177
Обеспечьте защиту в глубину	177
Никогда не доверяйте введенным данным	178
Соблюдайте принцип наименьшего уровня привилегий	178
Предполагайте, что внешние системы являются незащищенными	178
Сокращайте поверхность атаки	178
Отключайте ненужные средства	179
Защита приложения	179
Защита интранет-приложения	180
Аутентификация с помощью форм	184
Предохраниние против атак	191
Внедрение SQL-кода	192
Межсайтовые сценарии	196
Подделка межсайтовых запросов	197
Резюме	199
Глава 10. Разработка веб-приложений для мобильных устройств	200
Мобильные возможности ASP.NET MVC 4	200
Добавление мобильных возможностей в приложение	202
Создание мобильного представления для аукционных товаров	202
Начало работы с jQuery Mobile	203
Улучшение представления с помощью jQuery Mobile	205
Устранение настольных представлений на мобильном сайте	210
Совершенствование мобильного интерфейса	210
Адаптивная визуализация	210
Дескриптор окна просмотра	211
Определение мобильных функциональных возможностей	212
Медиа-запросы CSS	213
Представления, специфичные для браузера	214
Создание нового мобильного приложения с нуля	216
Сдвиг парадигмы в jQuery Mobile	216
Шаблон Mobile Application в ASP.NET MVC 4	216
Использование шаблона Mobile Application инфраструктуры ASP.NET MVC 4	218
Резюме	220

Часть III. Выход за стандартные рамки	221
Глава 11. Параллельные, асинхронные и операции над данными в реальном времени	222
Асинхронные контроллеры	222
Создание асинхронного контроллера	223
Обстоятельства, при которых используются асинхронные контроллеры	225
Асинхронные коммуникации реального времени	225
Сравнение моделей приложений	225
Опрос HTTP	226
Длительный опрос HTTP	227
События, отправляемые сервером	227
Веб-сокеты	228
Расширение возможностей коммуникаций реального времени	229
Конфигурирование и настройка	233
Резюме	234
Глава 12. Кеширование	235
Типы кеширования	235
Кеширование серверной стороны	235
Кеширование клиентской стороны	236
Технологии кеширования серверной стороны	236
Кеширование на уровне запросов	236
Кеширование на уровне пользователей	237
Кеширование на уровне приложений	238
Кеш ASP.NET	238
Кеш вывода	240
“Кеширование бублика”	243
“Кеширование дырки от бублика”	245
Распределенное кеширование	246
Технологии кеширования клиентской стороны	251
Кеш браузера	251
API-интерфейс ApplicationCache	252
Локальное хранилище	254
Резюме	256
Глава 13. Технологии оптимизации клиентской стороны	257
Структура страницы	257
Структура HTTP-запроса	258
Рекомендуемые приемы	259
Делайте меньше HTTP-запросов	260
Используйте сеть доставки контента	260
Добавляйте заголовок Expires или Cache-Control	260
Компоненты, сжатые с помощью GZip	263
Размещайте таблицы стилей в верхней части документа	265
Размещайте сценарии в нижней части документа	265
Делайте сценарии и стили внешними	266
Сокращайте поиск в DNS	267
Минимизируйте сценарии и стили	268

Избегайте перенаправлений	269
Удаляйте дублированные сценарии	270
Конфигурируйте теги ETag	271
Измерение производительности клиентской стороны	272
Ввод в работу ASP.NET MVC	274
Пакетирование и минимизация	275
Резюме	278
Глава 14. Расширенная маршрутизация	279
Нахождение пути	279
URL и поисковая оптимизация	281
Построение маршрутов	282
Стандартные и необязательные параметры маршрута	283
Порядок и приоритет маршрутизации	284
Маршрутизация на существующие файлы	285
Игнорирование маршрутов	285
Универсальные маршруты	286
Ограничения маршрутов	287
Исследование маршрутов с использованием Glimpse	289
Маршрутизация на основе атрибутов	289
Расширение маршрутизации	293
Конвейер маршрутизации	293
Резюме	297
Глава 15. Многократно используемые компоненты пользовательского интерфейса	298
Что ASP.NET MVC предлагает в готовом виде	298
Частичные представления	298
Расширения класса HtmlHelper или специальные вспомогательные методы HTML	299
Шаблоны отображения и редактирования	299
Html.RenderAction()	299
Продвижение на шаг вперед	300
Генератор одиночных файлов Razor	300
Создание многократно используемых представлений ASP.NET MVC	302
Создание многократно используемых вспомогательных методов ASP.NET MVC	305
Модульное тестирование представлений Razor	307
Резюме	309
Часть IV. Контроль качества	311
Глава 16. Регистрация в журнале	312
Обработка ошибок в ASP.NET MVC	312
Включение средства специальных ошибок	313
Обработка ошибок в действиях контроллеров	313
Определение глобальных обработчиков ошибок	314
Регистрация в журнале и трассировка	316
Регистрация ошибок в журнале	316
Мониторинг работоспособности ASP.NET	318
Резюме	321

Глава 17. Автоматизированное тестирование	322
Семантика тестирования	322
Ручное тестирование	323
Автоматизированное тестирование	324
Уровни автоматизированного тестирования	324
Модульные тесты	324
Интеграционные тесты	327
Приемочные тесты	328
Что собой представляет проект автоматизированных тестов?	329
Создание тестового проекта в Visual Studio	329
Создание и выполнение модульного теста	330
Тестирование приложения ASP.NET MVC	333
Тестирование модели	333
Разработка через тестирование	336
Написание чистых автоматизированных тестов	337
Тестирование контроллеров	339
Рефакторинг модульных тестов	342
Имитация зависимостей	342
Тестирование представлений	347
Покрытие кода	349
Миф о стопроцентном покрытии кода	351
Разработка поддающегося тестированию кода	351
Резюме	353
Глава 18. Автоматизация построения	354
Создание сценариев построения	355
Проекты Visual Studio являются сценариями построения	355
Добавление простой задачи построения	355
Выполнение построения	356
Возможности безграничны!	357
Автоматизация процесса построения	357
Типы автоматизированных построений	358
Создание автоматизированного построения	359
Непрерывная интеграция	362
Обнаружение проблем	362
Принципы непрерывной интеграции	363
Резюме	366
Часть V. Выход в свет	367
Глава 19. Развёртывание	368
Что необходимо развертывать	368
Основные файлы веб-сайта	368
Статический контент	370
Что не должно развертываться	370
Базы данных и другие внешние зависимости	371
Что требуется для приложения EBuу	372
Развёртывание на сервере IIS	372
Предварительные условия	373

Создание и конфигурирование веб-сайта IIS	373
Публикация из Visual Studio	375
Развертывание в Windows Azure	378
Создание учетной записи Windows Azure	378
Создание нового веб-сайта Windows Azure	379
Публикация веб-сайта Windows Azure через систему управления исходным кодом	380
Резюме	381
Часть VI. Приложения	383
Приложение А. Интеграция ASP.NET MVC и Web Forms	384
Выбор между ASP.NET MVC и ASP.NET Web Forms	384
Перевод сайта Web Forms на ASP.NET MVC	385
Добавление ASP.NET MVC к существующему приложению Web Forms	386
Копирование функциональности Web Forms в приложение ASP.NET MVC	388
Интеграция функциональности Web Forms и ASP.NET MVC	389
Управление пользователями	389
Управление кешем	389
Многое, многое другое!	389
Резюме	390
Приложение Б. Использование NuGet в качестве платформы	391
Установка инструмента командной строки NuGet	391
Создание пакетов NuGet	392
Файлы NuSpec	392
Генерация пакета NuGet из файла NuSpec	394
Структура пакета NuGet	395
Папка Content	395
Папка libs	396
Папка tools	397
Типы пакетов NuGet	397
Пакеты сборок	398
Пакеты инструментов	398
Метапакеты	398
Разделение пакетов NuGet	398
Публикация в открытом репозитории пакетов NuGet.org	398
Размещение собственного репозитория пакетов	399
Советы, трюки и ловушки	402
Ловушка: NuGet не решает проблемы “ада DLL”	402
Совет: используйте команду <code>Install-Package -Version</code> для установки специфичной версии пакета	403
Совет: используйте систему Semantic Versioning	404
Совет: помечайте “бета-пакеты” с помощью маркеров предварительной версии	404
Ловушка: избегайте указания “строгих” зависимостей от версий в файлах NuSpec	405
Совет: используйте специальные репозитории для управления версиями пакетов	406
Совет: сконфигурируйте свои построения непрерывной интеграции для генерации пакетов NuGet	407
Резюме	408

Приложение В. Рекомендуемые приемы	409
Используйте для управления зависимостями диспетчер пакетов NuGet	409
Полагайтесь на абстракции	409
Избегайте использования ключевого слова new	409
Избегайте прямых ссылок на объект HttpContext (используйте HttpContextBase)	410
Избегайте “магических строк”	410
Отдавайте предпочтение моделям перед словарем ViewData	410
Не записывайте HTML-разметку в “серверный” код	411
Не выполняйте бизнес-логику в представлениях	411
Консолидируйте часто используемые фрагменты представлений с помощью вспомогательных методов	411
Отдавайте предпочтение презентационным моделям перед прямым использованием бизнес-объектов	411
Инкапсулируйте операторы if во вспомогательные методы HTML в представлениях	411
Отдавайте предпочтение явным именам представлений	412
Отдавайте предпочтение объектам параметров перед длинными списками параметров	413
Инкапсулируйте разделяемую / общую функциональность, логику и данные с помощью фильтров действий или дочерних действий (Html.RenderAction)	414
Отдавайте предпочтение группированию действий в контроллеры на основе того, как они связаны с бизнес-концепциями	415
Избегайте группирования действий в контроллеры на основе технических отношений	415
Отдавайте предпочтение фильтрам действий на самом высоком подходящем уровне	415
Отдавайте предпочтение нескольким представлениям (и / или частичным представлениям) перед сложной логикой if-then-else, которая отображает и скрывает разделы	415
Отдавайте предпочтение шаблону Post-Redirect-Get при отправке данных формы	416
Отдавайте предпочтение задачам запуска перед логикой, размещаемой в методе Application_Start () (Global.asax)	417
Отдавайте предпочтение атрибуту авторизации перед императивными проверками безопасности	417
Отдавайте предпочтение использованию атрибута маршрута перед более обобщенными глобальными маршрутами	418
Подумайте об использовании маркера противодействия атакам CSRF	418
Подумайте об использовании атрибута AcceptVerbs для ограничения способов вызова действий	418
Подумайте об использовании кеширования вывода	419
Подумайте об удалении неиспользуемых механизмов представлений	419
Подумайте об использовании специальных результатов действий в уникальных сценариях	420
Подумайте об использовании асинхронных контроллеров для задач, которые могут выполняться параллельно	420
Приложение Г. Перекрестные ссылки: целевые темы, функциональные возможности и сценарии	421
Предметный указатель	424

Об авторах

Джесс Чедвик – независимый консультант по программному обеспечению, специализирующийся на веб-технологиях. Он обладает более чем десятилетним опытом разработки, начиная со встраиваемых устройств для мелких компаний и заканчивая веб-фермами масштаба предприятия для компаний из списка Fortune 500. Джесс имеет звания ASPIInsider, Microsoft MVP в ASP.NET и является заядлым членом сообщества, часто готовящим технические презентации, а также ведущим в составе группы пользователей NJDOTNET Central New Jersey .NET. Он проживает в Филадельфии, шт. Пенсильвания, со своей замечательной женой, дочерью и черным лабрадором.

Тодд Снайдер – архитектор программного обеспечения, консультант и лектор, специализирующийся на разработке многоуровневых и веб-приложений на платформе Microsoft. Он имеет более чем 18 лет опыта и в настоящее время работает в компании Infragistics главным консультантом, помогая заказчикам в достижении успеха.

Хришикеш Панда – разработчик настольных, веб- и мобильных приложений, обладающий более чем 14-летним опытом программирования.

Предисловие

Область, связанная с веб-приложениями, обширна и разнообразна. Инфраструктура Microsoft ASP.NET Framework, построенная поверх зрелой и надежной платформы .NET Framework, является одной из наиболее проверенных платформ в рамках всей отрасли. Инфраструктура ASP.NET MVC – это последнее дополнение от Microsoft мира ASP.NET, предоставляющее веб-разработчикам альтернативный подход к легкому построению веб-приложений.

Главная цель этой книги проста: помочь вам обрести полное понимание инфраструктуры ASP.NET MVC 4 Framework с самого начала. Тем не менее, дело на этом не заканчивается – фундаментальные концепции ASP.NET MVC сочетаются в книге с реальным пониманием, современными веб-технологиями (такими как HTML 5 и JavaScript-библиотека jQuery) и мощными архитектурными шаблонами, поэтому вы будете готовы к созданию не только веб-сайта, использующего ASP.NET MVC Framework, но надежного и масштабируемого веб-приложения, которое легко расширять и поддерживать в соответствии с растущими потребностями.

Предполагаемая читательская аудитория

Эта книга рассчитана на читателей, которые желают научиться применять Microsoft ASP.NET MVC Framework для построения надежных и легко сопровождаемых веб-сайтов. Хотя в книге используется множество примеров кода для детального описания этого процесса, она ориентирована не просто на разработчиков приложений. В большей части материала представлены концепции и технологии, которые принесут пользу как разработчикам, пишущим код приложения, так и руководителям, управляющим этими проектами разработки.

Что требуется для работы с этой книгой

Несмотря на то что эта книга призвана научить вас всему, что необходимо знать для создания надежных и легко сопровождаемых веб-приложений с помощью ASP.NET MVC Framework, в ней предполагается наличие у вас некоторых фундаментальных сведений о разработке приложений в Microsoft .NET Framework. Другими словами, вы должны уверенно пользоваться HTML, CSS и JavaScript для построения базовых веб-сайтов и обладать достаточным опытом работы с .NET Framework и языком C#, чтобы суметь создавать приложения уровня “Hello World”.



Примеры кода для этой книги доступны по адресу <https://github.com/ProgrammingAspNetMvcBook/CodeExamples>.

Соглашения, используемые в этой книге

В книге также приняты следующие типографские соглашения.

- **Курсив.** Применяется для новых терминов.
- **Моноширинный.** Применяется для листингов программ, а также внутри абзацев для ссылки на программные элементы, такие как имена переменных или функций, типы данных, переменные среды, операторы и ключевые слова.
- **Моноширинный полужирный.** Применяется для выделения раздела кода, а также команд или другого текста, который должен вводиться вручную пользователем.
- **Моноширинный курсив.** Применяется для выделения текста, который должен быть заменен значениями, предоставленными пользователем или определенным контекстом.



Здесь приводится совет, указание и общее замечание.



Здесь приводится предупреждение или предостережение.

Использование примеров кода

Данная книга призвана помочь вам в выполнении работы. В общем случае вы можете использовать приведенный здесь код в своих программах и документации. Вы не обязаны связываться с нами для получения специального разрешения, если только не воспроизводите значительную порцию кода. Например, написание программы, в которой встречаются многие фрагменты кода из этой книги, разрешения не требует. Однако продажа или распространение компакт-диска с примерами из книг O'Reilly требует разрешения. Ответ на вопрос путем ссылки на эту книгу и цитирования кода из примера разрешения не требует. Однако внедрение значительной части кода примера из этой книги в документацию по вашему продукту требует разрешения.

Мы высоко ценим указание авторства, хотя и не требуем этого. Например: “*Programming ASP.NET MVC 4* by Jess Chadwick, Todd Synder, and Hrusikesh Panda (O'Reilly). Copyright 2012 Jess Chadwick, Todd Synder, and Hrusikesh Panda, 978-1-449-32031-7”.

Если вам кажется, что использование вами примеров кода выходит за законные рамки или упомянутые выше разрешения, свяжитесь с нами по следующему адресу электронной почты: permissions@oreilly.com.

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо, либо просто посетить наш веб-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг.

Наши координаты:

E-mail: info@williamspublishing.com

WWW: <http://www.williamspublishing.com>

Информация для писем из:

России: 127055, г. Москва, ул. Лесная, д. 43, стр. 1

Украины: 03150, Киев, а/я 152

Начало работы

В этой части...

Глава 1. Основы ASP.NET MVC

Глава 2. ASP.NET MVC для разработчиков Web Forms

Глава 3. Работа с данными

Глава 4. Разработка на стороне клиента

Основы ASP.NET MVC

Microsoft ASP.NET MVC – это инфраструктура для разработки веб-приложений, построенная поверх зрелой и популярной платформы .NET Framework. Инфраструктура ASP.NET MVC в основном полагается на проверенные шаблоны и приемы разработки, которые делают акцент на слабо связанной архитектуре приложения и хорошо поддающемся сопровождению коде.

В этой главе мы рассмотрим основы, заложенные в ASP.NET MVC, начиная с происхождения и архитектурных концепций, на которых построена архитектура, и заканчивая использованием Microsoft Visual Studio 2011 для создания полностью функционирующего веб-приложения ASP.NET MVC. Затем мы погрузимся в детали проекта веб-приложения ASP.NET MVC и посмотрим, какие средства предоставляются инфраструктурой ASP.NET MVC с самого начала, включая работающую веб-страницу и встроенную аутентификацию с помощью форм, которая позволяет пользователям регистрироваться и входить на сайт.

К концу этой главы вы будете иметь не только функционирующее веб-приложение ASP.NET MVC, но также и обладать достаточным пониманием основ инфраструктуры ASP.NET MVC, чтобы незамедлительно приступить к построению приложений с ее помощью. Остальной материал этой книги просто построен на этих основах и показывает, как получить максимальный эффект от ASP.NET MVC Framework в любом веб-приложении.

Платформа веб-разработки от Microsoft

Понимание прошлого может оказать большую помощь в оценке настоящего; поэтому перед тем, как заняться анализом работы ASP.NET MVC, давайте посвятим некоторое время описанию происхождения этой инфраструктуры.

Много лет тому назад в Microsoft осознали потребность в платформе веб-разработки на основе Windows и активно занялись построением решения. За последние два десятилетия компания Microsoft предложила сообществу разработчиков несколько платформ веб-разработки.

Active Server Pages (ASP)

Первой платформой веб-разработки от Microsoft была Active Server Pages (ASP) – язык написания сценариев, в котором код и разметка размещались в одном файле, а каждый физический файл соответствовал странице на веб-сайте. Подход со сценариями на стороне сервера, предложенный ASP, стал очень популярным и многие веб-сайты приняли его. Некоторые из этих сайтов продолжают обслуживать посетителей и

в настоящее время. Тем не менее, разработчики желали большего. Они запрашивали такие возможности, как улучшенное многократное использование кода, усовершенствованное разделение ответственности и более простое применение принципов объектно-ориентированного программирования. В 2002 г. компания Microsoft предложила в качестве решения этих запросов технологию ASP.NET.

ASP.NET Web Forms

Как и в ASP, веб-сайты ASP.NET были основаны на страничном подходе, при котором каждая страница веб-сайта представлена в форме физического файла (называемого веб-формой (Web Form)) и доступна через имя этого файла. В отличие от страницы, использующей ASP, страница Web Forms обеспечивает некоторое разделение кода и разметки за счет разнесения веб-контента по двум разным файлам, один из которых предназначен для разметки, а другой – для кода. Технология ASP.NET и подход Web Forms обслуживали потребности разработчиков многие годы, и они продолжают выступать в качестве платформы веб-разработки у множества разработчиков приложений .NET. Тем не менее, некоторые разработчики для .NET считают подход Web Forms слишком абстрагированным от лежащих в основе HTML, JavaScript и CSS. Наверное, просто некоторым разработчикам невозможно угодить! Или все-таки можно?

ASP.NET MVC

В Microsoft быстро обнаружили в сообществе разработчиков ASP.NET растущую потребность в наличии чего-то, отличающегося от страничного подхода Web Forms, и в 2008 г. выпустили первую версию ASP.NET MVC. Представляя собой полное отступление от подхода Web Forms, ASP.NET MVC отбрасывает архитектуру на основе страниц, а вместо нее полагается на архитектуру *модель-представление-контроллер* (Model-View-Controller – MVC).



В отличие от инфраструктуры ASP.NET Web Forms, которая была введена как замена своему предшественнику ASP, ASP.NET MVC никоим образом не замещает существующую платформу Web Forms Framework. Скорее наоборот – приложения ASP.NET MVC и Web Forms построены на основе платформы ASP.NET Framework, которая предоставляет общий API-интерфейс для веб-разработки, интенсивно используемый обеими инфраструктурами.

Идея того, что ASP.NET MVC и Web Forms – это просто разные пути создания веб-сайта ASP.NET, является основополагающей в настоящей книге; более подробно эта концепция освещена в главе 2 и приложении А.

Архитектура “модель-представление-контроллер”

Шаблон “модель-представление-контроллер” – это архитектурный шаблон, который поддерживает строгую изоляцию между отдельными частями приложения. Такая изоляция более известна как *разделение ответственности* или, если пользоваться более общими терминами, как *слабое связывание*. Практически все аспекты MVC – и, следовательно, ASP.NET MVC Framework – управляются такой целью сохранения разнородных частей приложения изолированными друг от друга.

Проектирование архитектуры приложений в слабо связанном стиле привносит ряд как краткосрочных, так и долгосрочных преимуществ.

- **Разработка.** Отдельные компоненты не зависят напрямую от других компонентов, а это означает, что их более просто разрабатывать в изоляции. Компоненты также легко заменять или замещать, предотвращая возникновение сложностей в ситуациях, когда один компонент влияет на разработку других компонентов, с которыми он может взаимодействовать.
- **Тестируемость.** Слабое связывание компонентов позволяет применять тестовые реализации на месте производственных версий компонентов. Скажем, за счет замены компонента, выполняющего обращения к базе данных, компонентом, который просто возвращает статические данные, можно избежать взаимодействия с физической базой данных на этапе разработки. Способность компонентов легко меняться местами с пробными представлениями существенно упрощает процесс тестирования, который радикально увеличивает надежность системы с течением времени.
- **Сопровождение.** Изоляция компонентов означает, что изменения в логике обычно изолируются в небольшом числе компонентов — очень часто в одном. С учетом того, что степень влияния изменения обычно зависит от его масштаба, модификация меньшего количества компонентов — это однозначно хорошо!

Шаблон MVC разделяет приложение на три уровня: модель, представление и контроллер (рис. 1.1). Каждый из этих уровней выполняет очень специфичную работу, за которую он отвечает, и, что наиболее важно, не беспокоится о том, как свою работу делают другие уровни.

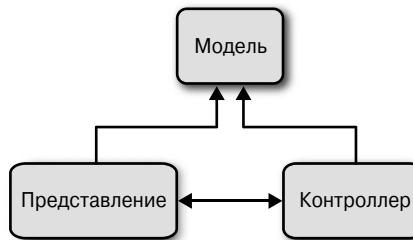


Рис. 1.1. Архитектура MVC

Модель

Модель представляет основную бизнес-логику и данные. Модель инкапсулирует свойства и поведение сущности предметной области и открывает свойства, которые описывают эту сущность. Например, класс `Auction` представляет в приложении концепцию “предмета на аукционе” и может открывать свойства наподобие `Title` (название) и `CurrentBid` (текущая ставка), а также поведение в форме методов вроде `Bid()`.

Представление

Представление отвечает за преобразование модели или моделей в визуальную презентацию. В веб-приложениях это чаще всего означает генерацию HTML-разметки для визуализации в браузере пользователя, хотя представления могут проявляться во многих формах. Например, одна и та же модель может быть визуализирована в виде HTML, PDF, XML или даже электронной таблицы.

Следуя принципу разделения ответственности, представления должны концентрироваться только на *отображении* данных и не могут содержать какую-либо бизнес-логику – бизнес-логика остается в модели, которая должна предоставлять представлению все, что необходимо.

Контроллер

Контроллер управляет логикой приложения и действует в качестве координатора между представлением и моделью. Контроллеры получают пользовательский ввод через представление и затем взаимодействуют с моделью для выполнения специфичных действий, передавая результаты обратно представлению.

Нововведения, появившиеся в версии ASP.NET MVC 4

В этой книге подробно исследуется инфраструктура ASP.NET MVC Framework и объясняется, как извлечь максимум из предлагаемых ею возможностей и функциональности. Несмотря на то что уже вышла четвертая версия инфраструктуры, в большей части книги рассматривается функциональность, которая существовала в предшествующих версиях. Если вы хорошо знаете предыдущие версии, можете пропускать то, что вам известно, и изучать только новые добавления.

Ниже приведены краткие описания новых возможностей версии ASP.NET MVC 4 вместе со ссылками на разделы книги, где они демонстрируются в действии.

- **Асинхронные контроллеры.** Веб-сервер Internet Information Server (IIS) обрабатывает каждый поступающий запрос в новом потоке, поэтому каждый новый запрос отнимает один из конечного числа потоков, доступных IIS, даже если поток запроса находится в состоянии ожидания (к примеру, ждет ответа от запроса, отправленного базе данных или веб-службе). И хотя последние обновления в .NET Framework 4.0 и IIS 7 существенно увеличили стандартное количество потоков, доступных пулу потоков IIS, по-прежнему рекомендуется избегать удержания системных ресурсов дольше, чем это необходимо. В версии ASP.NET MVC Framework 4 появились *асинхронные контроллеры*, которые позволяют эффективнее обрабатывать такие типы длительно выполняющихся запросов в более асинхронной манере. За счет использования асинхронных контроллеров можно сообщить инфраструктуре о необходимости освободить поток, который обрабатывает запрос, и позволить выполнять другие задачи по обработке, пока запрос ожидает завершения интересующих его задач. После их завершения инфраструктура вернется в состояние, в котором она покинула поток, и возвратит тот же самый ответ, как если бы запрос обрабатывался обычным синхронным контроллером – за исключением того, что становится возможной обработка намного большего количества запросов одновременно! Асинхронные контроллеры подробно рассматриваются в главе 11.
- **Режимы отображения.** Число подключенных к Интернету устройств постоянно растет, поэтому вы должны быть готовы предоставить им возможность просматривать ваш сайт. Очень часто данные, отображаемые на этих устройствах, совпадают с данными, отображаемыми на настольных компьютерах, с тем лишь отличием, что визуальные элементы должны учитывать меньший форм-фактор мобильных устройств. *Режимы отображения* ASP.NET MVC предоставляют простой, основанный на соглашениях подход к настройке представлений и компо-

новок, которые предназначены для различных устройств. В главе 10 будет показано, как применять на сайте режимы отображения в виде части комплексного подхода к добавлению поддержки мобильных устройств.

- **Пакетирование и минимизация.** Несмотря на широкую доступность в наши дни высокоскоростных подключений к Интернету, ваш сайт никогда не должен полагаться на это. На самом деле, если речь идет об общем времени загрузки, то потери даже долей секунды в нескольких местах могут накапливаться и в какой-то момент начнут весьма отрицательно влиять на воспринимаемую посетителями производительность сайта. Такие концепции, как комбинирование сценариев и таблиц стилей и минимизация, могут не выглядеть чем-то новым, тем не менее, в версии .NET Framework 4.5 они являются фундаментальной частью инфраструктуры. Более того, ASP.NET MVC включает в себя и расширяет ключевую функциональность платформы .NET Framework, чтобы сделать эти инструменты еще более удобными для использования в разрабатываемых приложениях ASP.NET MVC. В главе 13 эти концепции объясняются более подробно; там также показано, как применять новые инструменты, предлагаемые ASP.NET and ASP.NET MVC Framework.
- **Web API.** Простые службы данных HTTP быстро становятся основным способом поставки данных для постоянно растущего разнообразия приложений, устройств и платформ. Инфраструктура ASP.NET MVC всегда предоставляла возможность возврата данных в различных форматах, включая JSON и XML; однако *ASP.NET Web API* совершенствует это взаимодействие, предлагая более современную модель программирования, которая ориентируется на предоставление развитых служб данных, а не на действия контроллера, приводящие к возврату данных. В главе 6 вы увидите, как задействовать AJAX на стороне клиента, причем для этого будут использоваться службы ASP.NET Web API.

Знаете ли вы...?

Инфраструктура ASP.NET MVC поставляется с открытым исходным кодом! Да, это так – начиная с марта 2012 г., весь исходный код ASP.NET MVC, Web API и Web Pages Framework доступен для просмотра и загрузки на сайте CodePlex (<http://aspnetwebstack.codeplex.com>). Более того, разработчики могут создавать собственные ветви и даже отправлять исправления для основного исходного кода инфраструктуры!

Введение в EBuy

Эта книга призвана показать не только все тонкости ASP.NET MVC Framework, но также продемонстрировать использование инфраструктуры в реальных приложениях. Проблема, связанная с такими приложениями, заключается в том, что само понятие “реальные” указывает на определенный уровень сложности и уникальности, который не может быть адекватно представлен в единственном демонстрационном приложении.

Вместо того чтобы попытаться предоставить решение для каждой встречающейся задачи, мы сформировали список сценариев и проблем, с которыми чаще всего сталкивались сами и слышали от других разработчиков. Хотя этот список может и не включать абсолютно все сценарии, встречающиеся при построении того или иного приложения, мы уверены, что он представляет большинство реальных задач, с которыми многие разработчики имеют дело при создании своих приложений ASP.NET MVC.



Мы вовсе не шутим — мы действительно составили список, и он находится в конце этой книги! В приложении Г приведена таблица перекрестных ссылок для всех функциональных возможностей и сценариев, которые были рассмотрены в книге, с указанием главы (или глав), где их можно найти.

Для покрытия сценариев в этом списке мы построим веб-приложение, которое объединяет все сценарии в приложение, близкое к реальному, но в то же время ограничиваясь областью, понятной каждому: сайт онлайновых аукционов.

В приложении EBuу мы построим сайт онлайновых аукционов на ASP.NET MVC! С высокуюровневой точки зрения цели этого сайта предельно ясны и прямолинейны: позволить пользователям указать список товаров, которые они хотят продать, и предложить цены на товары, которые они хотят купить. Однако при более глубоком анализе становится понятно, что приложение несколько сложнее, нежели кажется на первый взгляд, поскольку требует применения не только всех возможностей, предлагаемых ASP.NET MVC, но также и интеграции с другими технологиями.

Тем не менее, EBuу — это не просто набор кода, сопровождаемый книгу. В каждой главе книги не только приводится описание дополнительных возможностей и функциональности, но также и демонстрируется их использование при построении приложения EBuу — от создания проекта и до развертывания, что позволит вам эффективно отслеживать весь процесс проектирования.



Полный исходный код приложения EBuу доступен для загрузки на веб-сайте книги по адресу <http://www.programmingaspnetmvc.com>.

А теперь перейдем от разговоров к делу, приступив к построению приложения.

Установка ASP.NET MVC

Перед началом разработки приложений ASP.NET MVC понадобится загрузить и установить ASP.NET MVC 4 Framework. Для этого всего лишь нужно зайти на веб-сайт ASP.NET MVC (<http://www.asp.net/mvc>) и щелкнуть на кнопке со словом **Install** (Установить).

В результате запустится установщик веб-платформы (Web Platform Installer), бесплатный инструмент, который упрощает установку многих веб-инструментов и приложений. Следуйте указаниям мастера Web Platform Installer для загрузки и установки инфраструктуры ASP.NET MVC 4 и ее зависимостей на своей машине.

Обратите внимание, что для успешной установки и использования ASP.NET MVC 4 на машине должны быть установлены, по меньшей мере, PowerShell 2.0 и Visual Studio 2010 Service Pack 1 или Visual Web Developer Express 2010 Service Pack 1. К счастью, если они еще не установлены, Web Platform Installer определяет это и обеспечивает загрузку и установку последних версий PowerShell и Visual Studio.



Если в настоящее время вы пользуетесь предыдущей версией ASP.NET MVC и желаете создавать приложения ASP.NET MVC 4, а также продолжать работу с приложениями ASP.NET MVC 3, то переживать не стоит. Новую версию ASP.NET MVC можно установить бок о бок с установленной версией ASP.NET MVC 3.

После того как все необходимое загружено и установлено, пора переходить к следующему шагу – созданию первого приложения ASP.NET MVC 4.

Создание приложения ASP.NET MVC

Установщик ASP.NET MVC 4 добавляет новый тип проекта Visual Studio под названием **ASP.NET MVC 4 Web Application** (Веб-приложение ASP.NET MVC 4). Это ваша точка входа в мир ASP.NET MVC и то, что вы будете использовать для создания нового проекта веб-приложения EBuy, которое строится на протяжении данной книги.

Для создания нового проекта выберите версию Visual C# шаблона ASP.NET MVC 4 Web Application и введите Ebuy .Website в поле Name (Имя), как показано на рис. 1.2.

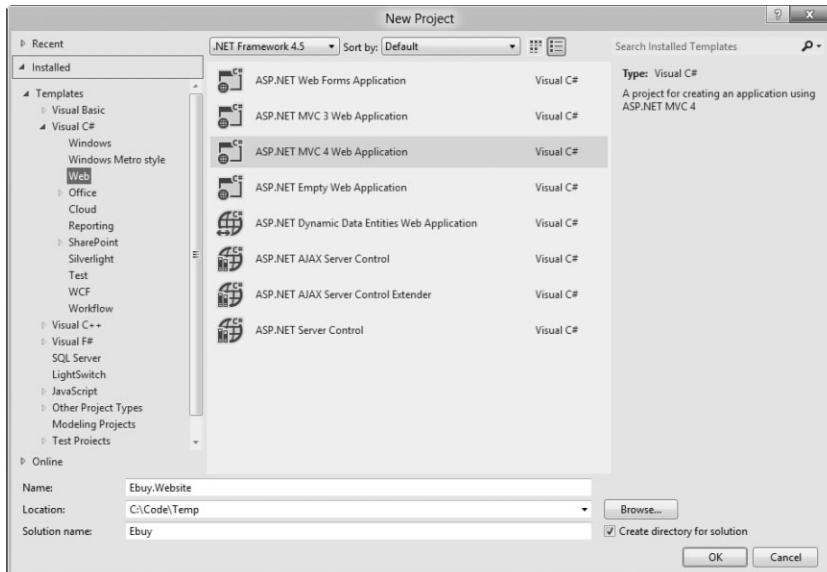


Рис. 1.2. Создание проекта EBuy

После щелчка на кнопке OK для продолжения откроется другое диалоговое окно с дополнительными опциями (рис. 1.3).

Это диалоговое окно позволяет настроить приложение ASP.NET MVC 4, которое среда Visual Studio генерирует, предоставляя возможность указать, какой вид сайта ASP.NET MVC необходимо создать.

Шаблоны проекта

Первым делом, ASP.NET MVC 4 предлагает несколько шаблонов проектов, каждый из которых ориентируется на отличающийся сценарий.

- **Empty.** Шаблон Empty (Пустое) создает пустое приложение ASP.NET MVC 4 с соответствующей структурой папок, которая включает ссылки на сборки ASP.NET MVC, а также на ряд библиотек JavaScript, которые могут использоваться по-путно. Этот шаблон также включает стандартную компоновку представления и генерирует файл Global.asax, включающий стандартный код конфигурации, который необходим большинству приложений ASP.NET MVC.

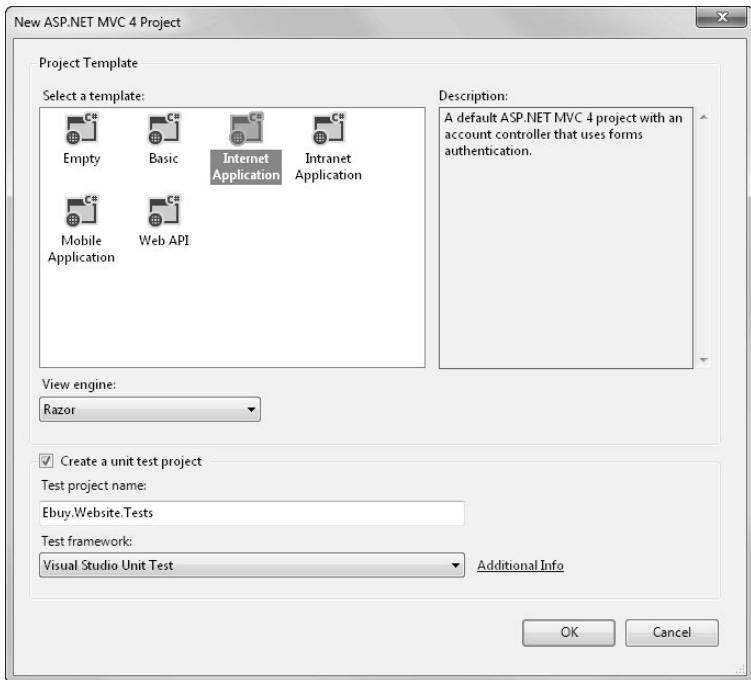


Рис. 1.3. Настройка проекта EBuy

- **Basic.** Шаблон Basic (Базовое) создает структуру папок, соответствующую соглашениям ASP.NET MVC 4, и включает ссылки на сборки ASP.NET MVC. Этот шаблон представляет абсолютный минимум, который необходим для того, чтобы приступить к созданию проекта ASP.NET MVC 4, но не более – далее всю работу придется выполнять самостоятельно!
- **Internet Application.** Шаблон Internet Application (Интернет-приложение) расширяет шаблон Empty за счет добавления простого стандартного контроллера (`HomeController`), контроллера `AccountController` со всей логикой, требуемой для регистрации и входа пользователей на веб-сайт, и стандартными представлениями для обоих контроллеров.
- **Intranet Application.** Шаблон Intranet Application (Инtranет-приложение) очень похож на шаблон Internet Application за исключением того, что он предварительно сконфигурирован для использования аутентификации Windows, которая желательна в сценариях с корпоративными сетями.
- **Mobile Application.** Шаблон Mobile Application (Мобильное приложение) – это другая вариация шаблона Internet Application. Однако данный шаблон оптимизирован для мобильных устройств и включает JavaScript-инфраструктуру `jQuery Mobile` и представления, применяющие HTML-разметку, которая лучше работает с `jQuery Mobile`.
- **Web API.** Шаблон Web API является еще одной вариацией шаблона Internet Application, включающей заранее сконфигурированный контроллер Web API. Интерфейс Web API – это новая облегченная инфраструктура веб-служб HTTP, поддерживающих REST, которая довольно хорошо интегрируется с ASP.NET MVC.

Интерфейс Web API представляет собой удобный вариант, когда нужно быстро и легко создать службы данных, которые будут потребляться приложениями, оснащенными AJAX. Этот новый API-интерфейс подробно рассматривается в главе 6.

Диалоговое окно New ASP.NET MVC Project (Новый проект ASP.NET MVC) также позволяет выбрать *механизм представлений*, или синтаксис, на котором будут записываться представления. Для построения образцового приложения ЕВу мы будем использовать новый синтаксис Razor, так что можно оставить в списке View engine (Механизм представлений) выбранным значение, предлагаемое по умолчанию (Razor). Имейте в виду, что механизм представлений приложения может быть изменен в любой момент; эта опция предназначена только для информирования мастера о том, какую разновидность представлений необходимо *генерировать*, а не для того, чтобы навсегда замкнуть приложение на специфичный механизм представлений.

Наконец, укажите, должен ли мастер генерировать проект модульного тестирования для этого приложения (с помощью флажка Create a unit test project (Создать проект модульного тестирования)). И снова не стоит особенно переживать по поводу этого выбора – как и во всех других решениях Visual Studio, проект модульного тестирования может быть добавлен к веб-приложению ASP.NET MVC в любое время.

Завершив выбор нужных опций, щелкните на кнопке OK, чтобы заставить мастер генерировать новый проект!

Управление пакетами NuGet

Если вы обратите внимание на строку состояния, когда Visual Studio создает новый проект веб-приложения, то можете заметить сообщения (вроде *Installing package AspNetMvc...* (Установка пакета AspNetMvc...)), касающиеся того факта, что шаблон проекта используется диспетчером пакетов NuGet для установки и управления ссылками на сборки в приложении. Концепция применения диспетчера пакетов для управления зависимостями приложения – особенно как часть фазы нового шаблона проекта – несомненно, очень мощная; она также является новой особенностью типов проектов ASP.NET MVC 4.

Появившийся в виде части установщика ASP.NET MVC 3, диспетчер NuGet предлагает альтернативный рабочий поток для управления зависимостями приложения. Хотя диспетчер NuGet в действительности не является частью ASP.NET MVC Framework, он выполняет немалую работу, чтобы сделать возможным построение ваших проектов.

Пакет NuGet может содержать смесь сборок, контента и даже инструментов, оказывающих помощь в разработке. Во время установки пакета диспетчер NuGet будет добавлять сборки в список References (Ссылки) целевого проекта, копировать любой контент в структуру папок приложения и регистрировать любые инструменты в текущем пути, позволяя их запускать в консоли диспетчера пакетов (Package Manager Console).

Тем не менее, самый важный аспект пакетов NuGet – и в действительности главная причина создания самого диспетчера NuGet – связан с управлением зависимостями. Приложения .NET не являются монолитными программами, включающими единственную сборку; на самом деле, большинство сборок в своей работе полагаются на ссылки на другие сборки. Более того, сборки обычно зависят от специфических версий (или, по меньшей мере, от минимальной версии) других сборок.

В сущности, NuGet просчитывает потенциально сложные отношения между всеми сборками, от которых зависит приложение, и обеспечивает наличие всех необходимых сборок с корректными их версиями.

Шлюзом ко всем возможностям NuGet является диспетчер пакетов NuGet (NuGet Package Manager). Получить доступ к диспетчеру пакетов NuGet можно двумя путями.

1. *Графический пользовательский интерфейс.* Диспетчер пакетов NuGet имеет графический пользовательский интерфейс, который упрощает поиск, установку, обновление и удаление пакетов для проекта. Для доступа к графическому интерфейсу диспетчера пакетов щелкните правой кнопкой мыши на проекте веб-сайта в окне Solution Explorer (Проводник решения) и выберите в контекстном меню пункт Manage NuGet Packages... (Управлять пакетами NuGet...).
2. *Консольный режим.* Консоль диспетчера библиотечных пакетов (Library Package Manager Console) – это окно Visual Studio, содержащее интегрированную командную строку PowerShell, которая специально сконфигурирована для доступа к диспетчеру библиотечных пакетов (Library Package Manager). Если эта консоль еще не открыта в Visual Studio, получить доступ к ней можно через пункт меню Tools⇒Library Package Manager⇒Package Manager Console (Сервис⇒Диспетчер библиотечных пакетов⇒Консоль диспетчера пакетов). Для установки пакета с помощью окна консоли диспетчера пакетов необходимо просто ввести команду Install-Package Имя_Пакета. Например, чтобы установить пакет Entity Framework, введите команду Install-Package EntityFramework. Консоль диспетчера пакетов загрузит пакет EntityFramework и установит его внутри текущего проекта. После этого сборки Entity Framework станут видимыми в списке References проекта.

Соглашение по конфигурации

Чтобы упростить разработку веб-сайтов и помочь разработчикам увеличить свою продуктивность, инфраструктура ASP.NET MVC, где только возможно, опирается на концепцию *соглашения по конфигурации* (convention over configuration; или *соглашения над конфигурацией*, если подчеркивать преимущество соглашения перед конфигурацией). Это означает, что вместо явных конфигурационных настроек ASP.NET MVC просто предполагает, что разработчики будут следовать определенным соглашениям при построении своих приложений.

Хорошим примером использования соглашения по конфигурации в инфраструктуре является структура папок проекта ASP.NET MVC (рис. 1.4). В проекте имеются три специальных папки, которые соответствуют элементам шаблона MVC: Controllers, Models и Views. Должно быть вполне очевидным, что именно содержится в каждой из этих папок.

Заглянув внутрь этих папок, вы найдете еще больше действующих соглашений. Например, папка Controllers не только содержит все классы контроллеров приложения, но имена этих классов контроллеров следуют соглашению о завершении суффиксом Controller. Инфраструктура использует это соглашение для регистрации контроллеров приложения при запуске и связывании их с соответствующими маршрутами.

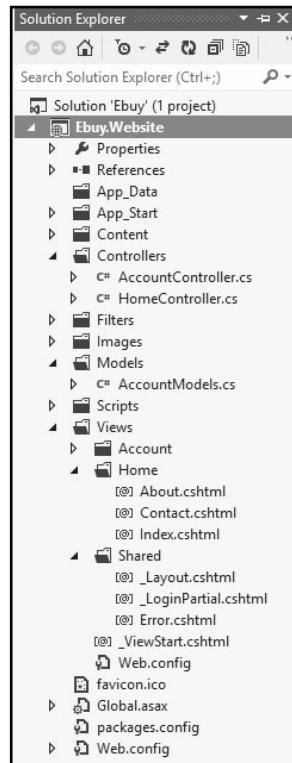


Рис. 1.4. Структура папок проекта ASP.NET MVC

А теперь загляните в папку Views. Кроме очевидного соглашения о том, что представления приложения должны находиться в данной папке, эта папка разделена на подпапки: подпапку Shared и дополнительные подпапки, содержащие представления для каждого контроллера. Такое соглашение помогает предотвратить указание разработчиками явных местоположений представлений, которые должны отображаться пользователям. Вместо этого разработчики могут просто задавать имя представления — скажем, Index — а инфраструктура постарается сделать все возможное, чтобы найти представление в папке Views, производя поиск сначала в подпапке, специфичной для контроллера, а затем в подпапке Shared.

На первый взгляд, концепция соглашения по конфигурации может выглядеть три-виальной. Тем не менее, эти, казалось бы, небольшие или малозначащие оптимизации могут обеспечить существенную экономию времени, улучшить читабельность кода и увеличить продуктивность труда разработчиков.

Запуск приложения

После того как процесс создан, можете нажать клавишу <F5>, чтобы запустить веб-сайт и посмотреть, что он визуализирует в браузере.

Поздравляем, вы только что создали свое первое приложение ASP.NET MVC 4!

Увидев результаты, отображаемые в браузере, могут возникнуть вопросы: “Что именно произошло? *Как это получилось?*”

На рис. 1.5 показано, как ASP.NET MVC обрабатывает запрос на высоком уровне.

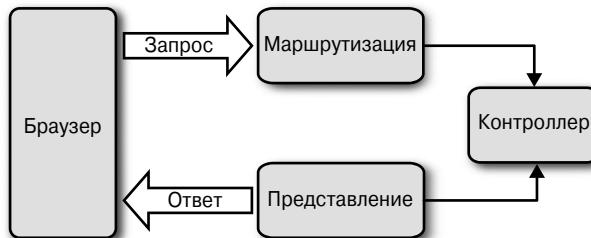


Рис. 1.5. Жизненный цикл запроса ASP.NET MVC

В оставшихся главах книги мы будем подробно рассматривать все компоненты этой диаграммы, а в следующих нескольких разделах приводятся пояснения данных фундаментальных строительных блоков ASP.NET MVC.

Маршрутизация

Весь трафик ASP.NET MVC начинается подобно трафику любого другого веб-сайта: с запроса какого-то URL. Это означает, что несмотря на отсутствие упоминания в названии, в основе каждого запроса ASP.NET MVC находится инфраструктура маршрутизации ASP.NET Routing.

Выражаясь простым языком, маршрутизация ASP.NET — это всего лишь система со-поставления с шаблоном. Во время запуска приложение регистрирует один или большее число шаблонов в *таблице маршрутов* инфраструктуры, тем самым сообщая системе маршрутизации о том, как поступать с любыми запросами, которые соответствуют этим шаблонам. Когда механизм маршрутизации получает запрос во время выполнения, он сопоставляет URL этого запроса с зарегистрированными шаблонами URL (рис. 1.6).

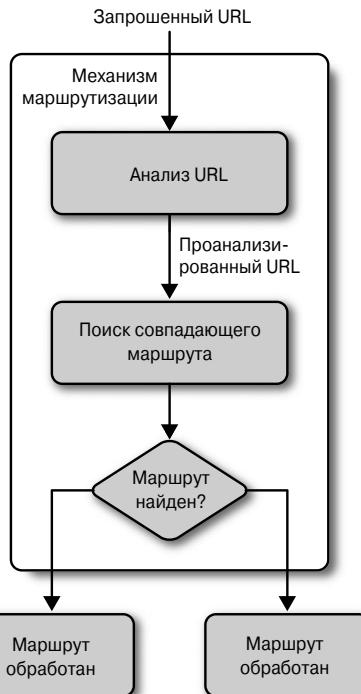


Рис. 1.6. Маршрутизация ASP.NET

Если механизм маршрутизации находит совпадающий шаблон в таблице маршрутизации, он перенаправляет запрос соответствующему обработчику для этого запроса.

В противном случае, если URL запроса не совпадает ни с одним из зарегистрированных шаблонов маршрутов, механизм маршрутизации указывает на то, что ему не удалось определить, как обрабатывать запрос, возвращая код состояния HTTP 404.

Конфигурирование маршрутов

Маршруты ASP.NET MVC отвечают за определение того, какой метод контроллера (по-другому известный как *действие контроллера*) выполнять для данного URL. С маршрутами связаны следующие свойства.

- **Уникальное имя.** Имя может использоваться в качестве специфичной ссылки на конкретный маршрут.
- **Шаблон URL.** Простой синтаксис шаблонов, который разбирает совпадающие URL на значащие сегменты.
- **Стандартные значения.** Необязательный набор стандартных значений для сегментов, определенных в шаблоне URL.
- **Ограничения.** Набор ограничений для применения к шаблону URL с целью более узкого определения URL, для которых он дает совпадение.

Стандартные шаблоны проектов ASP.NET MVC добавляют обобщенный маршрут, который использует приведенное ниже соглашение об URL для разбиения URL заданного запроса на три именованных сегмента, заключенных в фигурные скобки ({}): “контроллер”, “действие” и “идентификатор”:

{controller}/{action}/{id}

Этот шаблон маршрута регистрируется с помощью вызова расширяющего метода `MapRoute()` во время запуска приложения (в `App_Start/RouteConfig.cs`):

```
routes.MapRoute(  
    "Default", // Имя маршрута  
    "{controller}/{action}/{id}", // URL с параметрами  
    new { controller = "Home", action = "Index",  
          id = UrlParameter.Optional } // Стандартные значения параметров  
) ;
```

В дополнение к предоставлению имени и шаблона URL, этот маршрут также определяет набор стандартных параметров, предназначенных для использования в случае, если URL соответствует шаблону маршрута, но в действительности не указывает значения для каждого сегмента.

Например, в табл. 1.1 содержится список URL, которые совпадают с этим шаблоном маршрута, вместе с соответствующими значениями, которые инфраструктура маршрутизации предоставит каждому из них.

Таблица 1.1. Значения, устанавливаемые для URL, которые совпадают с шаблоном маршрута

URL	Контроллер	Действие	Идентификатор
/auctions/auction/1234	AuctionsController	Auction	1234
/auctions/recent	AuctionsController	Recent	
/auctions	AuctionsController	Index	
/	HomeController	Index	

Первый URL (`/auctions/auction/1234`) в табл. 1.1 обеспечивает полное соответствие, т.к. он удовлетворяет каждому сегменту шаблона маршрута. Однако по мере того, как в последующих URL убираются сегменты, начинают применяться стандартные значения для сегментов, для которых значения явно не указаны в URL.

Это очень важный пример того, как ASP.NET MVC использует концепцию соглашения по конфигурации: во время запуска приложения ASP.NET MVC обнаруживает все контроллеры приложения, выполняя в доступных сборках поиск классов, которые реализуют интерфейс `System.Web.Mvc.IController` (или являются производными от класса, реализующего этот интерфейс, такого как `System.Web.Mvc.Controller`) и имена которых заканчиваются суффиксом `Controller`. Когда инфраструктура маршрутизации применяет этот список для выяснения, к каким контроллерам она имеет доступ, суффикс `Controller` отбрасывается из всех имен классов контроллеров. Таким образом, для ссылки на контроллер должно использоваться сокращенное имя, например, на `AuctionsController` необходимо ссылаться как на `Auctions`, а на `HomeController` – как на `Home`.

Более того, значения контроллера и действия в маршруте не чувствительны к регистру символов. Это означает, что каждый из запросов `/Auctions/Recent`, `/auctions/Recent`, `/auctions/recent` или даже `/auCtIonS/rEcE_nt` будет успешно преобразован в действие `Recent` контроллера `AuctionsController`.



Шаблоны маршрутов URL являются относительными корня приложения, поэтому они не должны начинаться с косой черты (`/`) или указателя виртуального пути (`~/`). Шаблоны маршрутов, которые содержат эти символы, считаются недопустимыми и приведут к генерации исключения системы маршрутизации.

Как вы могли заметить, маршруты URL могут содержать больше информации, чем механизм маршрутизации способен извлечь. Тем не менее, для обработки запроса ASP.NET MVC механизм маршрутизации должен иметь возможность определять две критически порции информации: *контроллер* и *действие*. Затем механизм маршрутизации может передать эти значения исполняющей среде ASP.NET MVC для создания и выполнения указанного действия соответствующего контроллера.

Контроллеры

В контексте архитектурного шаблона MVC *контроллер* отвечает на пользовательский ввод (например, когда пользователь щелкает на кнопке Save (Сохранить)) и совместно работает с уровнями моделей, представлений и (довольно часто) доступа к данным. В приложении ASP.NET MVC контроллеры – это классы, содержащие методы, которые вызываются инфраструктурой маршрутизации для обработки запроса.

В качестве примера контроллера ASP.NET MVC рассмотрим класс HomeController, находящийся в файле Controllers/HomeController.cs:

```
using System.Web.Mvc;
namespace Ebuy.Website.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            // Страница описания приложения.
            ViewBag.Message = "Your app description page.";
            return View();
        }
        public ActionResult About()
        {
            // Страница краткого описания приложения.
            ViewBag.Message = "Your quintessential app description page.";
            return View();
        }
        public ActionResult Contact()
        {
            // Страница контактов.
            ViewBag.Message = "Your quintessential contact page.";
            return View();
        }
    }
}
```

Действия контроллеров

Как видите, классы контроллеров не являются чем-то особым; т.е. они не сильно по виду отличаются от других классов .NET. На самом деле всю обработку запросов выполняют *методы* классов контроллеров, которые называются *действиями контроллеров*.



Термины *контроллер* и *действие контроллера* часто будут использоваться взаимозаменяемым образом, даже в этой книге. Причина в том, что в шаблоне MVC между ними не делается никакой разницы. Тем не менее, инфраструктура ASP.NET MVC Framework больше связана с действиями контроллеров, поскольку они содержат действительную логику для обработки запроса.

Например, только что рассмотренный класс `HomeController` содержит три действия: `Index`, `About` и `Contact`. Таким образом, учитывая стандартный шаблон маршрута `{controller}/{action}/{id}`, когда производится запрос URL вида `/Home/About`, инфраструктура маршрутизации определяет, что этот запрос должен быть обработан методом `About()` класса `HomeController`. Затем ASP.NET MVC Framework создает новый экземпляр класса `HomeController` и выполняет его метод `About()`.

В этом случае метод `About()` очень прост: он передает данные представлению через свойство `ViewBag` (оно будет описано позже), после чего сообщает ASP.NET MVC Framework о необходимости отображения представления по имени `About` за счет вызова метода `View()`, который возвращает `ActionResult` типа `ViewResult`.

Результаты действий

Очень важно отметить, что работа контроллера заключается в уведомлении ASP.NET MVC Framework о том, *что* должно делаться следующим, но не *как* это должно делаться. Такое взаимодействие происходит с применением `ActionResult` – возвращаемых значений, которые, как ожидается, предоставляет каждое действие контроллера.

Например, когда контроллер решает отобразить представление, он сообщает ASP.NET MVC Framework об этом, возвращая `ViewResult`. Он не визуализирует само представление. Такое слабое связывание является еще одним хорошим примером реализации разделения ответственности (*что* делать против того, *каким образом* это делать).

Несмотря на тот факт, что каждое действие контроллера должно возвращать `ActionResult`, вы будете редко создавать их вручную. Вместо этого вы обычно будете пользоваться вспомогательными методами, предоставляемыми базовым классом `System.Web.Mvc.Controller`, которые кратко описаны ниже.

- `Content()`. Возвращает экземпляр `ContentResult`, который визуализирует произвольный текст, например, “Hello, world!”.
- `File()`. Возвращает экземпляр `FileResult`, который визуализирует содержимое файла, например, PDF.
- `NotFound()`. Возвращает экземпляр `NotFoundResult`, который визуализирует ответ в виде кода состояния HTTP 404.
- `JavaScript()`. Возвращает экземпляр `JavaScriptResult`, который визуализирует код JavaScript, например, `function hello() { alert(Hello, World!); }`.
- `Json()`. Возвращает экземпляр `JsonResult`, который сериализирует объект и визуализирует его в формате JSON (JavaScript Object Notation – нотация объектов JavaScript), например, `{ "Message": Hello, World! }`.
- `PartialView()`. Возвращает экземпляр `PartialViewResult`, который визуализирует только контент представления (т.е. представление без его разметки).
- `Redirect()`. Возвращает экземпляр `RedirectResult`, который визуализирует код состояния HTTP 302 (временное перенаправление) для перенаправления пользователя к заданному URL, например, `302 http://www.ebuy.com/auctions/recent`. Этот метод имеет родственный метод `RedirectPermanent()`, который также возвращает `RedirectResult`, но использует код состояния HTTP 301 для указания постоянного перенаправления, а не временного.

- `RedirectToAction()` и `RedirectToRoute()`. Функционируют подобно вспомогательному методу `Redirect()`, но только инфраструктура динамически определяет внешний URL, запрашивая механизм маршрутизации. Аналогично вспомогательному методу `Redirect()`, эти два метода также имеют варианты постоянного перенаправления: `RedirectToActionPermanent()` и `RedirectToRoutePermanent()`.
- `View()`. Возвращает экземпляр `ViewResult`, который визуализирует представление.

На основании этого списка можно сказать, что инфраструктура предоставляет результат действия практически для любой ситуации, которая должна поддерживаться, но если подходящего результата действия нет, можно создать собственный.



Хотя все действия контроллеров требуют предоставления экземпляра `ActionResult`, определяющего следующие шаги, которые должны быть предприняты для обработки запроса, не все действия контроллеров обязаны указывать `ActionResult` в качестве возвращаемого типа. Действия контроллеров могут применять любой возвращаемый тип, производный от `ActionResult`, или даже любой другой тип.

Когда инфраструктура ASP.NET MVC Framework сталкивается с действием контроллера, которое возвращает тип, отличный от `ActionResult`, она автоматически помещает значение в оболочку `ContentResult` и визуализирует его как низкоуровневый контент.

Параметры действий

Действия контроллеров подобны любым другим методам. На самом деле действие контроллера может даже указывать параметры, которые заполняются ASP.NET MVC с использованием информации из запроса, когда он обрабатывается. Эта функциональность называется *привязкой модели*, и она представляет собой одну из наиболее мощных и полезных возможностей ASP.NET MVC.

Перед тем как погружаться в детали работы привязки модели, давайте рассмотрим пример “традиционного” способа взаимодействия со значениями запроса:

```
public ActionResult Create()
{
    var auction = new Auction() {
        Title = Request["title"],
        CurrentPrice = Decimal.Parse(Request["currentPrice"]),
        StartTime = DateTime.Parse(Request["startTime"]),
        EndTime = DateTime.Parse(Request["endTime"]),
    };
    // ...
}
```

Действие контроллера в этом конкретном примере создает и заполняет свойства нового объекта `Auction` значениями, полученными прямо из запроса. Поскольку некоторые свойства `Auction` определены как разнообразные элементарные, отличные от `string` типы, действию также понадобится выполнить разбор каждого из соответствующих значений запроса в подходящий тип.

Этот пример может выглядеть простым и прямолинейным, но в действительности он очень хрупкий: в случае неудачи любой попытки разбора все действие завершится

сбоем. Переход к применению различных методов TryParse() может помочь избежать большинства исключений, но потребует написания дополнительного кода.

Побочным эффектом такого подхода является то, что каждое действие становится очень подробным. Недостаток написания подобного явного кода связан с тем, что на вас как разработчика возлагается бремя по выполнению всей работы каждый раз, когда это требуется. Большой объем кода также затеняет реальную цель: в настоящем примере это добавление нового объекта Auction в систему.

Основы привязки модели

Привязка модели не только позволяет избавиться от всего явного кода, она еще также очень проста в применении. Настолько проста, что на самом деле о ней даже думать не придется.

Например, ниже приведен то же самое действие контроллера, что и ранее, но на это раз с использованием параметров метода, привязанных к модели:

```
public ActionResult Create(
    string title, decimal currentPrice,
    DateTime startTime, DateTime endTime
)
{
    var auction = new Auction() {
        Title = title,
        CurrentPrice = currentPrice,
        StartTime = startTime,
        EndTime = endTime,
    };
    // ...
}
```

Теперь вместо явного извлечения значений из Request действие объявляет их как параметры. Когда инфраструктура ASP.NET MVC выполняет этот метод, она пытается заполнить параметры действия, используя те же самые значения из запроса, которые были показаны в предыдущем примере. Обратите внимание, что хотя мы не обращаемся к словарю Request напрямую, имена параметров по-прежнему очень важны, т.к. они соответствуют значениям из Request.

Тем не менее, объект Request – не единственное место, откуда связыватель модели получает значения. Инфраструктура производит поиск в различных местах, таких как данные маршрута, параметры строки запроса, значения отправленной формы и даже сериализированные объекты JSON. Например, в следующем фрагменте кода значение извлекается из URL просто за счет объявления параметра с тем же именем (пример 1.1).

Пример 1.1. Извлечение id из URL (вида /auctions/auction/123)

```
public ActionResult Auction(long id)
{
    var context = new EBuyContext();
    var auction = context.Auctions.FirstOrDefault(x => x.Id == id);
    return View("Auction", auction);
}
```



Где и как связыватель модели ASP.NET MVC ищет эти значения – в действительности является конфигурируемым и даже расширяемым. Подробное обсуждение привязки модели приводится в главе 8.

Как демонстрируют показанные примеры, привязка модели позволяет ASP.NET MVC поддерживать обычный шаблонный код, поэтому логика внутри действия может быть сосредоточена на предоставлении бизнес-значения. Оставшийся код является намного более значащим, не говоря уже о более высокой читабельности.

Привязка модели к сложным объектам

Применение подхода с привязкой модели даже к простым, элементарным типам может оказать большое влияние в плане выразительности кода. Однако в реальности дела обстоят намного сложнее – только самые базовые сценарии предполагают работу всего с парой параметров. К счастью, ASP.NET MVC поддерживает привязку как к сложным, так и к элементарным типам.

В следующем примере создается еще одна версия действия `Create`, на этот раз пропускающая промежуточные элементарные типы и осуществляющая привязку прямо к экземпляру `Auction`:

```
public ActionResult Create(Auction auction)
{
    // ...
}
```

Показанное здесь действие эквивалентно тому, что вы видели в предыдущем примере. Здесь все правильно: привязка модели к сложным объектам в ASP.NET MVC просто привела к устраниению *всего* шаблонного кода, требуемого для создания и заполнения нового экземпляра `Auction`! Этот пример демонстрирует реальную мощь привязки модели.

Фильтры действий

Фильтры действий предоставляют простой, но мощный способ модификации или расширения конвейера ASP.NET MVC за счет “внедрения” логики в определенных точках, помогая реализовать “сквозную функциональность”, которая применяется ко многим (или всем) компонентам приложения. Классическим примером сквозной функциональности является ведение журнала приложения, в равной степени примененного ко всем компонентам приложения вне зависимости от того, в чем заключается основная ответственность того или иного компонента.

Логика фильтров действий в основном вводится за счет применения атрибута `ActionFilterAttribute` к действию контроллера для оказания влияния на то, как выполняется это действие; это демонстрируется в следующем примере, где действие контроллера защищается от неавторизованного доступа путем применения `AuthorizeAttribute`:

```
[Authorize]
public ActionResult Profile()
{
    // Извлечь информацию профиля для текущего пользователя.
    return View();
}
```

Инфраструктура ASP.NET MVC Framework включает довольно много фильтров действий, ориентированных на общие сценарии. Эти фильтры действий будут использоваться повсеместно в книге, помогая решать разнообразные задачи чистым, слабо связанным способом.

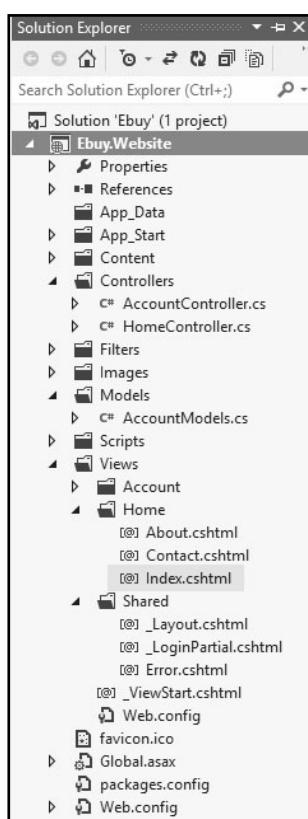


Фильтры действий – это великолепный путь применения специальной логики по всему сайту. Имейте в виду, что можно создавать также и собственные фильтры действий, расширяя базовый класс `ActionFilterAttribute` или любой класс фильтра действия ASP.NET MVC.

Представления

В ASP.NET MVC Framework действия контроллеров, которым необходимо отобразить HTML-разметку пользователю, возвращают экземпляр `ViewResult` – тип `ActionResult`, знающий, как визуализировать контент для ответа. Когда наступает время визуализации представления, ASP.NET MVC Framework ищет представление с использованием имени, предоставленного контроллером.

Взгляните на действие `Index` в `HomeController`:



Это действие вызывает вспомогательный метод `View()` для создания экземпляра `ViewResult`. Вызов `View()` без параметров, как в этом примере, указывает ASP.NET MVC на необходимость поиска представления с тем же именем, что и у текущего действия контроллера. В приведенном примере ASP.NET MVC будет искать представление `Index`, но где именно?

Определение местоположения представлений

Инфраструктура ASP.NET MVC полагается на соглашение, которое заключается в том, что все представления приложения хранятся внутри папки `Views` в корне веб-сайта. Более конкретно, ASP.NET MVC ожидает обнаружить представления в папках, именованных согласно контроллеру, к которому они относятся.

Таким образом, когда инфраструктура желает отобразить представление для действия `Index` в `HomeController`, она будет искать в папке `/Views/Home` файл по имени `Index`. На экранном снимке, показанном на рис. 1.7, видно, что шаблон проекта включил представление `Index.cshtml` автоматически.

Если найти представление с нужным именем в папке `Views` контроллера не удалось, ASP.NET MVC продолжает поиск в общей папке `/Views/Shared`.

Рис. 1.7. Определение местоположения представления Index



Папка `/Views/Shared` – великолепное место для размещения представлений, разделяемых между множеством контроллеров.

Теперь, когда найдено представление, запрошенное действием, откройте его и удостоверьтесь, что внутри там присутствует HTML-разметка и код. Однако это не просто любая HTML-разметка и код – это *Razor*!

Механизм Razor

Razor – это синтаксис, который позволяет комбинировать код и контент в плавной и выразительной манере. Несмотря на введение нескольких символов и ключевых слов, *Razor* не является новым языком. Вместо этого *Razor* позволяет писать код, используя известные вам языки, такие как C# или Visual Basic .NET.

Изучение *Razor* будет коротким, поскольку имеется возможность задействовать существующие знания, а не осваивать совершенно новый язык. Таким образом, если вы умеете разрабатывать код HTML и .NET на C# или Visual Basic .NET, то можете просто записать разметку вроде показанной ниже:

```
<div>This page rendered at @DateTime.Now</div>
```

Это даст следующий вывод:

```
<div>This page rendered at 1/29/2013 7:38:00 AM</div>
```

Приведенный пример начинается со стандартного HTML-дескриптора (`<div>`), за которым следует фрагмент “жестко закодированного” текста, порция динамического текста, визуализированная как результат ссылки на свойство .NET (`System.DateTime.Now`), и, наконец, закрывающий HTML-дескриптор (`</div>`).

Интеллектуальный анализатор *Razor* позволяет разработчикам более выразительно представлять логику и упрощает переходы между кодом и разметкой. Хотя синтаксис *Razor* может отличаться от других синтаксисов разметки (таких как Web Forms), он, в конечном счете, преследует ту же самую цель: визуализацию HTML.

Для иллюстрации этого аспекта взгляните на приведенные ниже фрагменты, которые демонстрируют примеры общих сценариев, реализованных с помощью разметки *Razor* и Web Forms.

Вот как выглядит оператор `if/else` в синтаксисе Web Forms:

```
<% if(User.IsAuthenticated) { %>
    <span>Hello, <%: User.Username %>!</span>
<% } %>
<% else { %>
    <span>Please <%: Html.ActionLink("log in") %></span>
<% } %>
```

А вот он же, но в синтаксисе *Razor*:

```
@if(User.IsAuthenticated) {
    <span>Hello, @User.Username!</span>
} else {
    <span>Please @Html.ActionLink("log in")</span>
}
```

Цикл `foreach` в рамках синтаксиса Web Forms может быть записан так:

```
<ul>
<% foreach( var auction in auctions) { %>
```

```
<li><a href="<%: auction.Href %>"><%: auction.Title %></a></li>
<% } %>
</ul>
```

В синтаксисе Razor он представляется следующим образом:

```
<ul>
@foreach( var auction in auctions ) {
    <li><a href="@auction.Href">@auction.Title</a></li>
}
</ul>
```

Несмотря на применение разного синтаксиса, оба фрагмента для каждого примера визуализируются в одну и ту же HTML-разметку.

Различие кода и разметки

Razor предоставляет два способа различения кода и разметки: фрагменты кода и блоки кода.

Фрагменты кода

Фрагменты кода (*code nuggets*) – это простые выражения, которые оцениваются и визуализируются встроенным образом. Они могут смешиваться с текстом и выглядят примерно так:

```
Not Logged In: @Html.ActionLink("Login", "Login")
```

Выражение начинается непосредственно после символа @, и механизм Razor достаточно интеллектуален, чтобы трактовать закрывающую круглую скобку как указатель конца этого конкретного оператора.

Показанный выше пример визуализирует следующий вывод:

```
Not Logged In: <a href="/Login">Login</a>
```

Обратите внимание, что фрагменты кода должны всегда возвращать разметку для визуализации представлением. Если вы напишете фрагмент кода, который оценивается как значение void, то получите ошибку при выполнении представления.

Блоки кода

Блок кода (*code block*) – это раздел представления, который содержит строго код, а не комбинацию разметки и кода. Блоки кода в Razor определяются как любой раздел шаблона Razor, помещенный внутрь символов @{ }. Символы @{ помечают начало блока, за которым следует любое количество строк полноценно сформированного кода. Символ } закрывает блок кода.

Имейте в виду, что код внутри блока кода не похож на код внутри фрагмента кода. Это обычный код, который должен следовать правилам текущего языка. Например, каждый оператор кода на языке C# должен завершаться точкой с запятой (;), как если бы он находился внутри класса в файле .cs.

Ниже приведен пример типичного блока кода:

```
@{
    LayoutPage = "~/Views/Shared/_Layout.cshtml";
    View.Title = "Auction " + Model.Title;
}
```

Блоки кода ничего не визуализируют для представления. Вместо этого они позволяют записывать произвольный код, не требующий возврата значений.

Кроме того, переменные, которые определены в блоках кода, могут использовать-ся фрагментами кода, находящимися в той же области видимости. Это значит, что переменные, определенные внутри цикла `foreach` или аналогичного контейнера, будут доступны только в рамках этого контейнера. С другой стороны, переменные, которые определены на верхнем уровне представления (за пределами любого вида контейнера), будут доступны любым другим блокам кода или фрагментам кода в том же представлении.

Чтобы прояснить сказанное, рассмотрим представление с несколькими переменными, определенными в различных областях видимости:

```
@{  
    // Переменные title и bids доступны по всему представлению.  
    var title = Model.Title;  
    var bids = Model.Bids;  
}  
<h1>@title</h1>  
<div class="items">  
    <!-- Цикл по объектам в переменной bids --&gt;<br/>    @foreach(var bid in bids) {  
        <!-- Переменная bid доступна только внутри цикла foreach --&gt;<br/>        <div class="bid">  
            <span class="bidder">@bid.Username</span>  
            <span class="amount">@bid.Amount</span>  
        </div>  
    }  
    <!-- Это приведет к ошибке: переменная bid не существует в этой<br/>        области видимости! -->  
    <div>Last Bid Amount: @bid.Amount</div>  
    </div>
```

Блоки кода являются средством для выполнения кода внутри шаблона и ничего не визуализируют для представления. В противоположность тому, как фрагменты кода должны обеспечивать возвращаемое значение для визуализации представлением, представление полностью игнорирует значения, возвращаемые блоком кода.

Компоновки

Механизм Razor предлагает возможность поддержки согласованного внешнего вида по всему сайту посредством *компонентов*. В этом случае единственное представление действует в качестве шаблона для всех других используемых представлений, определяя станичную компоновку и стиль на уровне целого сайта.

Шаблон компоновки обычно включает главную разметку (сценарии, таблицы стилей CSS и структурированные HTML-элементы, такие как контейнеры навигации и контента), указывающую местоположения внутри компоновки, в которых представления могут определять контент. Каждое представление на сайте затем ссылается на эту компоновку, добавляя со своей стороны только контент в заданные местоположения.

Взгляните на базовый файл компоновки Razor (`_Layout.cshtml`):

```
<!DOCTYPE html>  
<html lang="en">  
    <head>  
        <meta charset="utf-8" />  
        <title>@View.Title</title>  
    </head>
```

```
<body>
    <div class="header">
        @RenderSection("Header")
    </div>
    @RenderBody()
    <div class="footer">
        @RenderSection("Footer")
    </div>
</body>
</html>
```

Этот файл компоновки содержит главный HTML-контент, определяющий структуру HTML-разметки для всего сайта. Для взаимодействия с отдельными представлениями в компоновке используются переменные (такие как `@View.Title`) и вспомогательные функции наподобие `@RenderSection([ИмяРаздела])` и `@RenderBody()`.

После определения компоновки Razor представления ссылаются на эту компоновку и поставляют контент для ее разделов.

Ниже приведена базовая страница контента, которая ссылается на ранее определенный файл `_Layout.cshtml`:

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

@section Header {
    <h1>EBuy Online Auction Site<h1>
}

@section Footer {
    Copyright @DateTime.Now.Year
}

<div class="main">
    This is the main content.
</div>
```

Компоновки Razor и зависящие от них представления контента собираются вместе подобно фрагментам головоломки, каждый из которых определяет одну или более порций целой страницы. В результате сбора всех частей получается завершенная веб-страница.

Частичные представления

Хотя компоновки предлагают удобный способ многократного использования порций разметки и поддерживают согласованный внешний вид во множестве страниц на сайте, некоторые сценарии требуют более целенаправленного подхода.

Наиболее распространенный сценарий связан с необходимостью отображения одной и той же высокоуровневой информации во множестве местоположений сайта, но только на нескольких специфичных страницах, причем на каждой из них в отличающихся местах.

Например, сайт онлайновых аукционов Ebucu может визуализировать список кратких сведений о продаваемых предметах (содержащих только название, текущую цену и, возможно, миниатюрное изображение предмета) во многих местах на сайте, среди которых страница результатов поиска и список популярных предметов на домашней странице сайта.

Инфраструктура ASP.NET MVC поддерживает сценарии подобного рода через частичные представления.

Частичные представления – это такие представления, которые содержат целевую разметку, предназначенную для визуализации в виде части более крупного представления. В следующем фрагменте демонстрируется частичное представление для отображения кратких сведений о продаваемых предметах, которые упоминались выше:

```
@model Auction



<a href="@Model.Url">
        
    </a>
    <h4><a href="@Model.Url">@Model.Title</a></h4>
    <p>Current Price: @Model.CurrentPrice</p>


```

Чтобы визуализировать этот фрагмент как частичное представление, просто сохраните его в отдельном файле представления (например, /Views/Shared/Auction.cshtml) и воспользуйтесь одним из вспомогательных методов HTML инфраструктуры ASP.NET MVC – `Html.Partial()` – для визуализации представления в виде части другого представления.

Рассмотрим сказанное в действии. Ниже показан фрагмент, в котором производится проход по коллекции объектов, продающихся на аукционе, с применением приведенного выше частичного представления для визуализации HTML-разметки по каждому предмету:

```
@model IEnumerable<Auction>



## 


```

Обратите внимание, что первым параметром вспомогательного метода `Html.Partial()` является строка, содержащая имя представления без расширения.

Причина в том, что вспомогательный метод `Html.Partial()` – всего лишь простой уровень, определенный поверх мощного механизма представлений ASP.NET MVC. Данный метод визуализирует представление почти так же, как это происходит в ситуации, когда действие контроллера вызывает метод `View()` для возврата `ActionResult` типа `ViewResult`: механизм использует имя представления для нахождения и выполнения соответствующего представления.

При таком подходе частичные представления разрабатываются и выполняются в основном подобно любой другой разновидности представлений. Единственное отличие состоит в том, что они предназначены для визуализации в качестве части более крупного представления.

Второй параметр (`auction` в рассмотренном выше примере) принимает модель частичного представления, в частности как параметр модели во вспомогательном методе контроллера `View` (ИмяПредставления, [Модель]). Этот второй параметр модели является необязательным; если он не указан, то по умолчанию будет применяться модель представления, из которого был вызван метод `Html.Partial()`. Например, если в предыдущем примере второй параметр `auction` опустить, ASP.NET MVC будет передавать на его месте значение свойства `Model` (типа `IEnumerable<Auction>`) представления.



В приведенных выше примерах было показано, каким образом частичные представления могут организовывать многократно используемые разделы разметки, помогая сократить дублирование и снизить сложность представлений.

Это всего лишь один полезный способ применения частичных представлений; в разделе “Частичная визуализация” главы 6 будет показано, как с помощью частичных представлений обеспечить простой и эффективный способ расширения сайта возможностями AJAX.

Отображение данных

Архитектура MVC зависит от модели, представления и контроллера, которые все остаются отдельными, но работают вместе для достижения общей цели. В этом отношении задача контроллера — выступать в качестве “регулятора движения”, координируя разнообразные части системы для выполнения логики приложения. Результатом такой обработки обычно является определенный вид данных, которые должны быть переданы пользователю. Тем не менее, отображение их пользователю не входит в ответственность контроллера — для этого предназначены представления. В таком случае возникает вопрос: каким образом контроллер передает эту информацию представлению?

Инфраструктура ASP.NET MVC предлагает два способа передачи данных в рамках границ “модель-представление-контроллер”: ViewData и TempData. Эти объекты являются словарями, доступными в виде свойств в контроллерах и представлениях. Таким образом, передача данных из контроллера в представление сводится к установке значения в контроллере, как показано в следующем фрагменте кода из HomeController.cs:

```
public ActionResult About()
{
    ViewData["Username"] = User.Identity.Username;
    ViewData["CompanyName"] = "EBuy: The ASP.NET MVC Demo Site";
    ViewData["CompanyDescription"] =
        "EBuy is the world leader in ASP.NET MVC demoing!";
    return View("About");
}
```

и последующей ссылке на значение в представлении, что видно в части кода, взятой из файла About.cshtml:

```
<h1>@ViewData["CompanyName"]</h1>
<div>@ViewData["CompanyDescription"]</div>
```

Более чистый доступ к значениям ViewData через ViewBag

Контроллеры и представления ASP.NET MVC, открывающие свойство ViewData, также предоставляют доступ к похожему свойству по имени ViewBag. Свойство ViewBag — это просто оболочка вокруг ViewData, которая открывает словарь ViewData как объект dynamic.

Например, любые ссылки на значения в словаре ViewData из предыдущего фрагмента можно заменить ссылками на свойства dynamic объекта ViewBag:

```
public ActionResult About()
{
    ViewBag.Username = User.Identity.Username;
    ViewBag.CompanyName = "EBuy: The ASP.NET MVC Demo Site";
    ViewBag.CompanyDescription = "EBuy is the world leader in ASP.NET MVC demoing!";
    return View("About");
}
```

и:

```
<h1>@ViewBag.CompanyName</h1>
<div>@ViewBag.CompanyDescription</div>
```

Модели представлений

В дополнение к базовому поведению словаря, объект ViewData также предлагает свойство Model, которое представляет главный объект, являющийся целью запроса. Хотя свойство ViewData.Model концептуально ничем не отличается от ViewData["Model"], оно превращает модель в объект первого класса и считает ее более важной, чем другие данные, которые могут быть в запросе.

Например, предыдущие два фрагмента показали, что значения словаря CompanyName и CompanyDescription очевидно связаны друг с другом и представляют собой хорошую возможность быть помещенными вместе в модель.

Взгляните на CompanyInfo.cs:

```
public class CompanyInfo
{
    public string Name { get; set; }
    public string Description { get; set; }
}
```

действие About в HomeController.cs:

```
public ActionResult About()
{
    ViewBag.Username = User.Identity.Username;
    var company = new CompanyInfo {
        Name = "EBuy: The ASP.NET MVC Demo Site",
        Description = "EBuy is the world leader in ASP.NET MVC demoing!",
    };
    return View("About", company);
}
```

и фрагмент из About.cshtml:

```
@{ var company = (CompanyInfo)ViewData.Model; }
<h1>@company.Name</h1>
<div>@company.Description</div>
```

В этих фрагментах ссылки на значения словаря CompanyName и CompanyDescription объединены в экземпляре нового класса по имени CompanyInfo (company). Модифицированный фрагмент HomeController.cs также демонстрирует работу перегруженной версии вспомогательного метода View(). Эта перегруженная версия по-прежнему принимает в первом параметре имя желаемого представления. Однако вторым параметром является объект, который будет присвоен свойству ViewData.Model.

Теперь вместо установки значений словаря напрямую экземпляр company передается в качестве параметра model вспомогательному методу View() и представление (About.cshtml) может получить локальную ссылку на объект company для доступа к его значениям.

Строго типизированные представления

По умолчанию свойство Model, доступное внутри представлений Razor, является динамическим (dynamic), а это означает, что получать доступ к его значениям можно, не зная точный тип.

Тем не менее, учитывая статическую природу языка C# и великолепную поддержку средства IntelliSense для представлений Razor в Visual Studio, часто удобнее указывать тип модели страницы явным образом.

К счастью, Razor позволяет делать это очень просто — нужно лишь воспользоваться ключевым словом `@model` для указания имени типа модели:

```
@model Auction  
<h1>@Model.Name</h1>  
<div>@Model.Description</div>
```

Этот фрагмент модифицирует предыдущий пример `Auction.cshtml`, устранивая необходимость в добавлении промежуточной переменной для приведения `ViewData.Model`. Вместо этого в первой строке с помощью ключевого слова `@model` указывается, что типом модели является `CompanyInfo`, и это делает все ссылки на `ViewData.Model` строго типизированными и доступными напрямую.

Вспомогательные методы HTML и URL

Основная цель большинства веб-запросов заключается в доставке HTML-разметки пользователю, и ASP.NET MVC оказывает помощь в создании HTML. Помимо языка разметки Razor, инфраструктура ASP.NET MVC также предлагает множество вспомогательных классов и методов для простой и эффективной генерации HTML-разметки. Двумя самыми важными вспомогательными классами являются `HtmlHelper` и `UrlHelper`, которые доступны в контроллерах и представлениях через свойства `Html` и `Url`.

Вот пример применения этих двух вспомогательных классов:

```
<img src='@Url.Content("~/Content/images/header.jpg")' />  
@Html.ActionLink("Homepage", "Index", "Home")
```

Визуализированная разметка выглядит следующим образом:

```
<img src='/vdir/Content/images/header.jpg' />  
<a href="/vdir/Home/Index">Homepage</a>
```

По большей части, типы `HtmlHelper` и `UrlHelper` не имеют особо много собственных методов, а служат просто прослойкой, к которой инфраструктура подключает поведения через расширяющие методы. Это делает их важной точкой расширения, и вы будете встречать ссылки на указанные два типа повсеместно в книге.

В данный момент следует также отметить один момент: класс `HtmlHelper` помогает генерировать HTML-разметку, а класс `UrlHelper` оказывает помощь в генерации URL. Имейте это в виду и обращайтесь к упомянутым вспомогательным классам всегда, когда требуется генерировать URL или HTML.

Модели

Теперь, когда рассмотрены контроллеры и представления, самое время завершить определение MVC обсуждением *моделей*, которые обычно считаются наиболее важной частью архитектуры MVC. Но если они настолько важны, то почему они объясняются последними? В первую очередь потому, что уровень моделей существенно сложнее, т.к. он содержит всю бизнес-логику приложения — и эта логика отличается от приложения к приложению.

С технической точки зрения модель обычно состоит из нормальных классов, которые открывают доступ к данным в виде свойств и к логике в виде методов. Эти классы

бывают всех видов и размеров, но наиболее распространенным примером является “модель данных” или “модель предметной области”, основная работа которой заключается в управлении данными.

Например, взгляните на следующий фрагмент, в котором показан класс Auction — модель, управляющая всем образцовым приложением EBuу:

```
public class Auction
{
    public long Id { get; set; }
    public string Title { get; set; }
    public string Description { get; set; }
    public decimal StartPrice { get; set; }
    public decimal CurrentPrice { get; set; }
    public DateTime StartTime { get; set; }
    public DateTime EndTime { get; set; }
}
```

Хотя мы будем добавлять в класс Auction различную функциональность, такую как проверка достоверности и другое поведение, этот фрагмент все равно хорошо представляет модель в том, что определяет данные, которые образуют “предмет на аукционе”.

После того как мы построим класс Auction, понаблюдайте за другими классами (такими как службы и вспомогательные классы), которые все вместе формируют понятие модели в MVC.

Собираем все вместе

К этому моменту мы описали все части, образующие приложение ASP.NET MVC, но обсуждение было сосредоточено на коде, который Visual Studio генерирует в качестве части шаблона проекта. Другими словами, в действительности мы пока еще ничего не сделали. Так давайте изменим это!

В этом разделе внимание будет сосредоточено на том, как реализовать ту или иную возможность с нуля, создавая все, что необходимо для сценария примера: отображение предмета, продаваемого на аукционе. Как вы помните, каждый запрос ASP.NET MVC требует, по меньшей мере, трех вещей: маршрута, действия контроллера и представления (и необязательно модели).

Маршрут

Для выяснения шаблона маршрутизации, который необходимо применять для данного средства, потребуется сначала определить, как должен выглядеть URL для доступа к этому средству. В рассматриваемом примере мы собираемся избрать относительно стандартный URL вида `Auctions/Details/[Идентификатор предмета]`; например, <http://www.ebuy.biz/Auctions/Details/1234>.

Приятный сюрприз состоит в том, что стандартная конфигурация маршрутов уже поддерживает такой URL!

Контроллер

Далее нам понадобится создать контроллер для хранения всех действий, которые будут обрабатывать запрос.

Поскольку контроллеры — это просто классы, реализующие интерфейс контроллера ASP.NET MVC, вы могли бы вручную добавить в папку `Controllers` новый класс, производный от `System.Web.Mvc.Controller`, и приступить к добавлению действий

контроллера в этот класс. Тем не менее, среда Visual Studio предлагает несколько инструментов, которые выполняют большую часть работы по созданию новых контроллеров. Для этого просто щелкните правой кнопкой мыши на папке **Controllers** и выберите в контекстном меню пункт **Add Controller...** (Добавить Контроллер...); в результате откроется диалоговое окно **Add Controller** (Добавить контроллер), показанное на рис. 1.8.

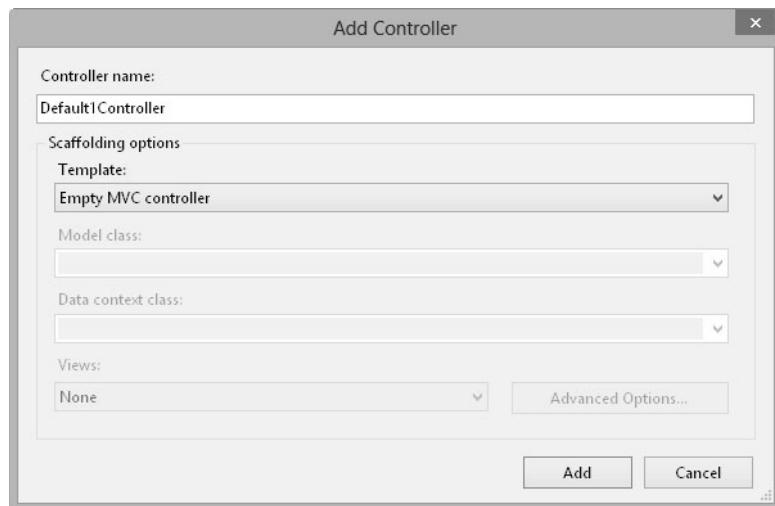


Рис. 1.8. Добавление контроллера в приложение ASP.NET MVC

В диалоговом окне **Add Controller** сначала необходимо ввести имя нового класса контроллера (в этом случае `AuctionsController`), а затем используемый шаблон для формирования. Кроме того, предоставляется набор опций, позволяющих в некоторой степени управлять тем, как ASP.NET MVC будет генерировать новый класс контроллера.

Шаблоны контроллера

Диалоговое окно **Add Controller** предлагает несколько других шаблонов контроллеров, которые могут помочь в более быстрой разработке контроллеров (все они перечислены ниже).

Шаблон Empty MVC controller

Предлагаемый по умолчанию шаблон **Empty MVC controller** (Пустой контроллер MVC) является самым простым из всех. По причине своей простоты он не поддерживает никаких опций настройки. Этот шаблон всего лишь создает новый контроллер с заданным именем, который имеет единственное сгенерированное действие `Index`.

Шаблон MVC controller with read/write actions and views, using Entity Framework

Шаблон **MVC controller with read/write actions and views, using Entity Framework** (Контроллер MVC с действиями чтения/записи и представлениями, использующий Entity Framework) является довольно впечатляющим. Он начинается с того же вывода, что и описанный ниже шаблон **MVC controller with empty read/write actions** (Контроллер MVC с пустыми действиями чтения/записи), но вдобавок генерирует код, который

помогает получать доступ к объектам, хранящимся в контексте Entity Framework, и даже генерирует представления Create (Создание), Edit (Редактирование), Details (Подробности) и Delete (Удалить) для этих объектов. Этот шаблон послужит хорошим быстрым стартом в ситуации, когда в проекте для доступа к данным используется инфраструктура Entity Framework, и в некоторых случаях сгенерированного им кода может оказаться вполне достаточно для поддержки операций чтения, редактирования, обновления и удаления данных.

Шаблон MVC controller with empty read/write actions

Шаблон MVC controller with empty read/write actions (Контроллер MVC с пустыми действиями чтения/записи) генерирует тот же код, что и шаблон Empty MVC controller (Пустой контроллер MVC), но добавляет несколько дополнительных действий, которые, скорее всего, понадобятся для обеспечения стандартных операций “доступа к данным”: Details, Create, Edit и Delete.

Шаблоны контроллера Web API

Последние три шаблона – Empty API controller (Пустой контроллер Web API), API controller with empty read/write actions (Контроллер Web API с пустыми действиями чтения/записи) и API controller with read/write actions and views, using Entity Framework (Контроллер Web API с действиями чтения/записи и представлениями, использующий Entity Framework) – являются эквивалентами Web API шаблонов контроллеров MVC с аналогичными названиями. Мы рассмотрим эти шаблоны более подробно при обсуждении функциональности Web API инфраструктуры ASP.NET MVC в главе 6.



Относительно кода контроллеров, генерируемого Visual Studio, интересно отметить, что действия Index и Details имеют только по одному методу, тогда как действия Create, Edit и Delete определяют по две перегруженных версии метода, одна из которых декорирована атрибутом `HttpPostAttribute`, а другая – нет.

Причина в том, что действия Create, Edit и Delete для своего завершения требуют два запроса: первый запрос возвращает представление, с которым пользователь может взаимодействовать для создания второго запроса, действительно выполняющего желаемое действие (создание, редактирование либо удаление данных). Это весьма распространенное взаимодействие в веб, и в книге вы увидите многочисленные примеры его использования.

К сожалению, мы пока еще не можем применять Entity Framework, поэтому сейчас необходимо выбрать вариант MVC controller with empty read/write actions и щелкнуть на кнопке Add (Добавить) для запуска генерации средой Visual Studio следующего класса контроллера.

После того как Visual Studio завершит создание класса `AuctionsController`, отыщите действие `Details` и модифицируйте его так, чтобы оно создавало новый экземпляр показанной ранее модели `Auction` и передавало этот экземпляр представлению, используя метод `View(object model)`.

Да, это сильно упрощенный пример. Обычно такая информация откуда-то извлекается, скажем, из базы данных, и мы покажем, как это делать, в главе 4, но в рассматриваемом примере мы применим жестко закодированные значения:

```
public ActionResult Details(long id = 0)
{
```

```

var auction = new Ebuy.Website.Models.Auction {
    Id = id,
    Title = "Brand new Widget 2.0",
    Description = "This is a brand new version 2.0 Widget!",
    StartPrice = 1.00m,
    CurrentPrice = 13.40m,
    StartTime = DateTime.Parse("6-15-2012 12:34 PM"),
    EndTime = DateTime.Parse("6-23-2012 12:34 PM"),
};

return View(auction);
}

```

Представление

Имея в наличии действие контроллера Details, а также возможность передачи данных представлению, самое время заняться созданием этого представления.

Как и с классом контроллера в предыдущем разделе, можно вручную добавить новые представления (и папки для их хранения) непосредственно в папку Views; однако если вы предпочитаете некоторую автоматизацию, то Visual Studio предлагает еще один мастер для создания представлений и соответствующих папок.

Чтобы добавить представление с применением мастера Visual Studio, щелкните правой кнопкой мыши где-нибудь внутри кода действия в контроллере и выберите в контекстном меню пункт Add View (Добавить представление); в результате отобразится диалоговое окно Add View (Добавление представления), показанное на рис. 1.9. Оно очень похоже на диалоговое окно Add Controller, которое использовалось для генерации AuctionsController.

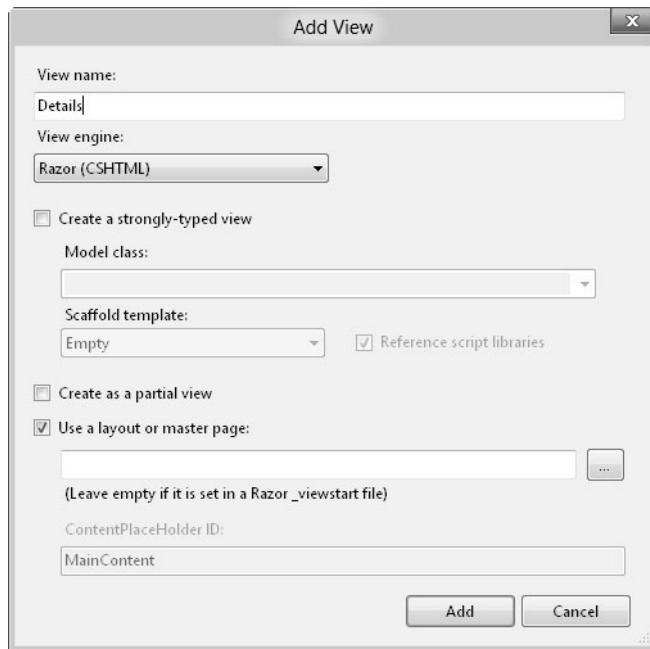


Рис. 1.9. Добавление представления в приложение ASP.NET MVC

Диалоговое окно Add View начинается с запроса имени для нового представления, которым по умолчанию будет имя действия контроллера, откуда было открыто это диалоговое окно (например, Details, если из действия Details). Затем диалоговое окно позволяет выбрать синтаксис (в списке View engine (Механизм представлений)), который необходимо использовать при написании кода представления. По умолчанию устанавливается синтаксис, выбранный при создании веб-проекта, однако (как упоминалось ранее) имеется возможность переключения между синтаксисами: например, для некоторых представлений можно применять синтаксис Razor, а для всех остальных – синтаксис ASPX Web Forms.

Как и в диалоговом окне Add Controller, остальные опции в окне Add View касаются того, что делать с кодом и разметкой, которые будет генерировать Visual Studio при создании нового представления. Например, можно указать на создание строго типизированного представления (как обсуждалось в разделе “Строго типизированные представления” ранее в этой главе), выбрав тип модели в списке классов проекта или вручную набрав имя типа. Если выбрано строго типизированное представление, мастер также позволит выбрать шаблон (такой как Edit, Create, Delete), который будет анализировать тип модели и генерировать для него подходящие поля формы.

Это отличный способ быстро получить код, сэкономив на наборе, поэтому воспользуемся им, отметив флажок Create a strongly typed view (Создать строго типизированное представление), выбрав модель Auction в раскрывающемся списке Model class (Класс модели) и указав Details в списке Scaffold template (Шаблон для формирования).



Среда Visual Studio включает в список Model class только классы, которые были успешно скомпилированы, поэтому если вы не видите ранее созданный класс Auction, попробуйте скомпилировать решение и затем откройте диалоговое окно Add View заново.

Наконец, среде Visual Studio понадобится сообщить, является это представление частичным или должно ссылаться на какую-то компоновку. Когда для написания страниц используется синтаксис ASPX Web Forms и отмечен флажок Create as a partial view (Создать как частичное представление), Visual Studio создаст представление как пользовательский элемент управления (User Control; .ascx), а не как полную страницу (.aspx). Однако в случае применения синтаксиса Razor отметка флажка Create as a partial view дает весьма незначительный эффект – Visual Studio будет создавать файл одного и того же типа (.cshtml или .vbhtml) как для частичных представлений, так и для полных страниц. При использовании синтаксиса Razor этот флажок оказывает воздействие только на разметку, генерируемую внутри нового представления.

Для целей этой демонстрации можно сохранить стандартные установки: флажок Create as a partial view оставить не отмеченым, флажок Use a layout or master page (Использовать компоновку или мастер-страницу) – отмеченым, а поле под ним – пустым (рис. 1.10).

Когда все готово, щелкните на кнопке Add (Добавить) и Visual Studio добавит новое представление в проект. После завершения вы увидите, что среда Visual Studio проанализировала модель Auction и сгенерировала требуемую HTML-разметку – снабженную ссылками на вспомогательные методы HTML, такие как `Html.DisplayFor` – для отображения всех полей Auction.

В этот момент вы должны иметь возможность запустить сайт, перейти к действию контроллера (например, /auctions/details/1234) и увидеть подробности объекта Auction, визуализированные в браузере, как показано на рис. 1.11.

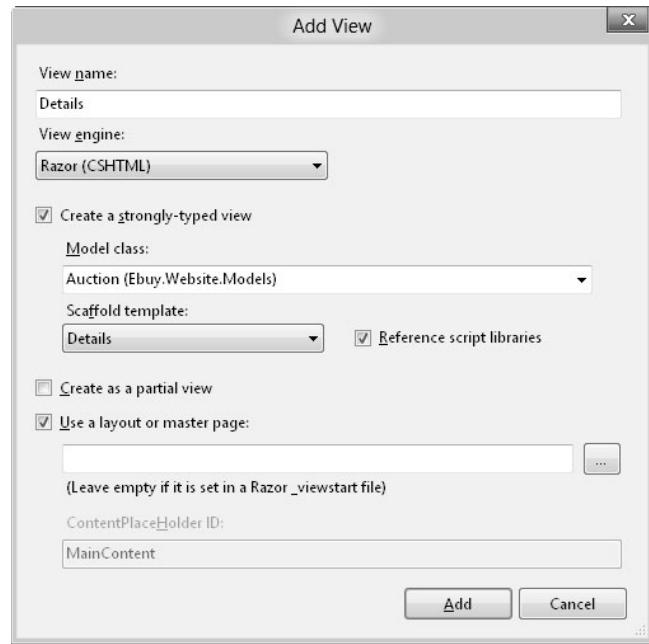


Рис. 1.10. Настройка нового представления

Помните, что HTML-разметка, которую генерирует Visual Studio, является лишь отправной точкой, помогающей сэкономить время. После генерации можете свободно изменять разметку любым желаемым образом.

Поздравляем, вы только что создали свое первое действие контроллера и представление с нуля!

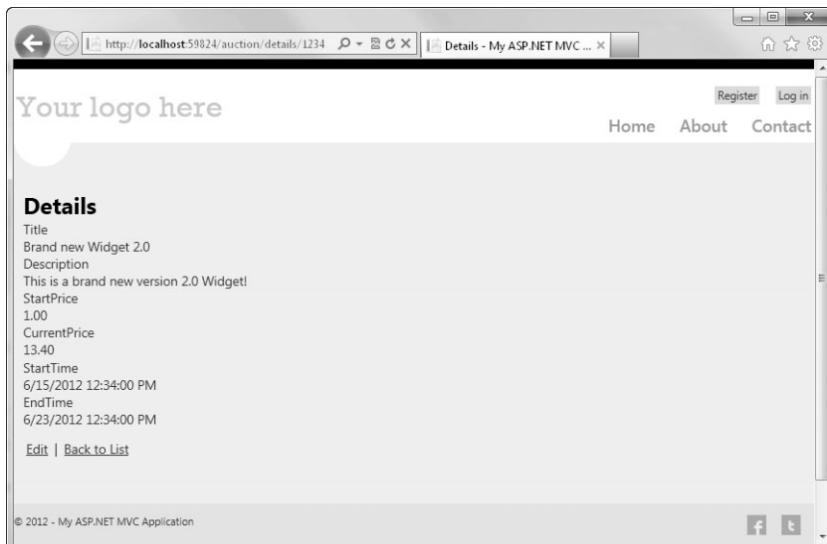


Рис. 1.11. Новое представление, визуализированное в браузере

Аутентификация

До сих пор мы рассматривали только те аспекты, которые касались создания приложения ASP.NET MVC. Однако есть еще одна очень важная концепция, с которой необходимо ознакомиться перед изучением дальнейших материалов: как защитить сайт, требуя от пользователей аутентификации, прежде чем они смогут получить доступ к определенным действиям контроллеров.

Вы могли заметить, что шаблон `Internet Application` генерирует контроллер `AccountController` (вместе с несколькими поддерживающими представлениями), который предлагает готовую полную реализацию аутентификации с помощью форм. Это признание того, что безопасность и аутентификация имеют критически важное значение почти во всех приложениях, и в какой-то момент вы, возможно, решите ограничить доступ к части или ко всему веб-приложению, разрешив его только заданным пользователям (или группам пользователей) и запретив неавторизованный доступ со стороны остальных посетителей. Таким образом, учитывая, что аутентификация нужна в любом случае, почему бы Visual Studio не генерировать такой контроллер и представления с самого начала?

Традиционная тактика, используемая для ограничения возможностей приложений ASP.NET, заключается в применении параметров авторизации к специфичным страницам или каталогам через параметры конфигурации в файле `web.config`. К сожалению, такой подход не работает с приложениями ASP.NET MVC, потому что эти приложения опираются на маршрутизацию к действиям контроллеров, а не на физические страницы.

Вместо этого инфраструктура ASP.NET MVC Framework предоставляет атрибут `AuthorizeAttribute`, который может применяться к отдельным действиям контроллера — или даже к целым контроллерам — для ограничения доступа только аутентифицированным пользователям либо, в качестве альтернативы, конкретными пользователями и пользовательскими ролями.

Взгляните на действие `Profile` контроллера `UsersController`, создаваемого далее в книге, которое отображает информацию профиля для текущего пользователя:

```
public class UsersController
{
    public ActionResult Profile()
    {
        var user = _repository.GetUserByUsername(User.Identity.Name);
        return View("Profile", user);
    }
}
```

Очевидно, что это действие даст сбой, если пользователь не вошел в систему. Применение атрибута `AuthorizeAttribute` к этому действию контроллера обеспечивает отклонение любых запросов со стороны пользователей, не прошедших аутентификацию:

```
public class UsersController
{
    [Authorize]
    public ActionResult Profile()
    {
        var user = _repository.GetUserByUsername(User.Identity.Name);
        return View("Profile", user);
    }
}
```

Для случаев, когда требуется большая избирательность в отношении пользователей, которые должны иметь доступ к действию контроллера, атрибут `AuthorizeAttribute` предлагает свойство `Users`, принимающее разделяемый запятыми список приемлемых имен пользователей, а также свойство `Roles`, принимающее список разрешенных ролей.

Если теперь неаутентифицированные пользователи попытаются обратиться к этому URL, они будут перенаправлены на URL входа: действие `Login` контроллера `AccountController`.

Класс `AccountController`

Чтобы оказать помощь в построении приложения, шаблон проекта `Internet Application` включает класс `AccountController`, который содержит действия контроллера, использующие поставщики членства ASP.NET (ASP.NET Membership Providers).

Класс `AccountController` предлагает довольно много готовой функциональности, а также представления для поддержки каждого действия контроллера. Это означает, что совершенно новое приложение ASP.NET MVC уже содержит следующие полностью реализованные возможности, не требующие никакого кодирования с вашей стороны:

- вход в систему;
- выход из системы;
- регистрация нового пользователя;
- изменение пароля.

Таким образом, в случае применения атрибута `AuthorizeAttribute` к любому действию контроллера пользователи перенаправляются на существующую страницу входа, автоматически созданную шаблоном проекта (рис. 1.12).

Когда пользователь пожелает создать новую учетную запись для последующего использования при входе, он может щелкнуть на ссылке `Register` (Регистрация) и перейти на готовую страницу регистрации (рис. 1.13).

Стандартная страница входа (Log in) имеет следующий вид:

Log in.

Use a local account to log in.

User name

Password

Remember me?

Log in

[Register](#) if you don't have an account.

Рис. 1.12. Стандартная страница входа

The screenshot shows a registration form with the following fields:

- User name**: A text input field containing "user1234".
- Password**: A text input field containing "****".
- Confirm password**: A text input field containing "****".

Below the inputs is a **Register** button.

At the top of the form, there are two error messages:

- The Password must be at least 6 characters long.
- The password and confirmation password do not match.

Рис. 1.13. Стандартная страница регистрации

Если готовые представления не подходят, их легко настроить с целью удовлетворения существующим требованиям.

Как было показано в этом разделе, помимо того, что инфраструктура ASP.NET MVC Framework упрощает защиту действий контроллеров, стандартный шаблон проекта реализует практически все, что необходимо пользователям для аутентификации на сайте.

Резюме

Инфраструктура ASP.NET MVC использует проверенный временем архитектурный шаблон “модель-представление-контроллер” (Model-View-Controller) для предоставления платформы разработки веб-сайтов, которые поддерживают архитектуру со слабым связыванием и многие другие популярные объектно-ориентированные шаблоны и приемы.

Инфраструктура ASP.NET MVC Framework с самого начала предлагает удобные шаблоны проектов и подход “соглашения по конфигурации”, который сокращает объем конфигурации, требуемой для создания и сопровождения приложений, и способствует возрастанию продуктивности процесса их разработки и внедрения.

Настоящая глава служила введением в фундаментальные концепции и базовые навыки, которые необходимы для того, чтобы приступить к построению приложений ASP.NET MVC 4. В оставшейся части книги этот фундамент будет расширен за счет демонстрации дополнительных возможностей, предлагаемых ASP.NET MVC Framework для создания надежных и легко сопровождаемых веб-приложений с использованием архитектурного шаблона MVC.

Так чего же вы ждете? Продолжайте чтение и узнайте все, что вам необходимо для построения самых лучших из всех веб-приложений, какие вы когда-либо писали!

ASP.NET MVC для разработчиков Web Forms

Несмотря на значительные отличия архитектурных подходов технологий ASP.NET MVC и Web Forms, в действительности у них много общего. В конце концов, обе они построены на основе ключевых API-интерфейсов ASP.NET и платформы .NET Framework. Таким образом, если вы являетесь разработчиком Web Forms, желающим изучить ASP.NET MVC Framework, то уже знаете довольно много.

В этой главе мы сравним ASP.NET MVC и Web Forms Frameworks, чтобы показать, насколько много концепций, используемых при построении приложений Web Forms, связаны с приемами, применяемыми в рамках ASP.NET MVC. Обратите внимание, что эта глава предназначена для разработчиков, которые хорошо знакомы с Web Forms Framework и хотят перенести свои знания на ASP.NET MVC, обеспечив более быстрый старт. Если же вы не очень хорошо знаете Web Forms Framework, можете пропустить эту главу и перейти к изучению остального материала книги.

Все это просто ASP.NET

Возможно, вы этого не знаете, но инфраструктура, используемая вами для разработки веб-страниц в .NET Framework – то, что вы, скорее всего, называете ASP.NET – в действительности может быть разделена на две части: визуальные компоненты пользовательского интерфейса (также известные как Web Forms) и невизуальные “серверные” веб-компоненты (также известные как ASP.NET). Эти две части проще всего определить по их пространствам имен .NET: все, что находится в пространствах имен `System.Web.UI.*`, может рассматриваться как Web Forms, а все, что определено в пространствах имен `System.Web.*` – как ASP.NET.

Подобно Web Forms, инфраструктура ASP.NET MVC (классы которой находятся в пространстве имен `System.Web.Mvc.*`) построена на строгих основах платформы ASP.NET. Как таковые, эти две инфраструктуры могут быть как очень похожи, так и крайне отличаться, в зависимости от того, под каким углом на них смотреть. В этом разделе рассматриваются сходные черты обеих инфраструктур, а в оставшемся материале главы – их многочисленные отличия.

Инструменты, языки и API-интерфейсы

Для начала следует отметить, что обе инфраструктуры имеют доступ ко всей платформе .NET Framework и всему тому, что она предлагает. Естественно, для взаимодействия с .NET Framework обе инфраструктуры опираются на .NET-языки C# и Visual

Basic .NET и могут обращаться к скомпилированным сборкам, разработанным на любом другом языке, который поддерживается .NET Framework.

В качестве приятного побочного эффекта эта возможность обеспечивает высокую степень многократного использования для существующего кода ASP.NET. Например, если имеется существующее приложение Web Forms, которое обращается к данным, хранящимся в XML-файлах, с помощью API-интерфейса `System.Xml`, скорее всего, будет возможность повторно использовать этот код в приложении ASP.NET MVC с небольшими модификациями либо вовсе без них.

Также не должно стать сюрпризом то, что для редактирования веб-сайтов ASP.NET MVC и проектов, от которых они зависят, будет применяться Visual Studio, в частности как для приложений Web Forms (или любых других .NET-приложений). Вы можете даже заметить несколько общих артефактов, таких как `web.config` и `Global.asax`, которые играют важную роль в приложениях ASP.NET MVC и Web Forms.

Обработчики и модули HTTP

Пожалуй, наиболее заметными частями .NET Framework, разделяемыми ASP.NET MVC и Web Forms, являются *обработчики HTTP* и *модули HTTP*. И хотя большинство классов API-интерфейса Web Forms (из пространства имен `System.Web.UI`) просто не будут работать в приложении ASP.NET MVC, обработчики и модули HTTP в действительности являются частью основного API-интерфейса ASP.NET (`System.Web`), поэтому они по-прежнему вполне функциональны в контексте приложения ASP.NET MVC. Фактически сам конвейер ASP.NET MVC начинается с обработки входящих запросов посредством обработчика HTTP!



Обязательно прочтите обо всех отличиях между ASP.NET MVC и Web Forms в этой главе и посмотрите, как они применяются к обработчикам и модулям HTTP. Хотя сами обработчики и модули HTTP функционируют хорошо в приложении ASP.NET MVC, имейте в виду, что существуют определенные аспекты, такие как состояние представления (View State), которые не работают так, как ожидается.

Управление состоянием

Управление состоянием, т.е. данными, связанными с пользователем – это важная часть любого приложения, и не поддерживающая состояния природа веб особенно усложняет решение этой задачи в веб-приложениях.

Для реализации управления состоянием в ASP.NET Web Forms применяется механизм *состояния представления* (View State), при котором данные состояния для запроса сериализируются и сохраняются в скрытом поле формы, отправляемом обратно серверу с каждым последующим запросом. Состояние представления является настолько критически важной частью платформы Web Forms, что практически каждая страница и компонент полагаются на него в определенной точке. Тем не менее, абстракция состояния представления также характеризуется своими недостатками, не последним из которых является тот факт, что каждый запрос, содержащий состояние представления, должен быть отправкой формы, а потенциальный размер данных в скрытом поле – данных, которые зачастую могут быть даже не востребованы – мог становиться довольно большим в случае некорректного использования.

Как объясняется в разделе “Управление состоянием” далее в главе, подход к управлению состоянием, принятый в ASP.NET MVC, значительно отличается, и в большинстве случаев ответственность за реализацию управления состоянием возлагается на

разработчика. Однако более важно то, что инфраструктура ASP.NET MVC полностью отбрасывает управление состоянием.

К счастью, в дополнение к состоянию представления платформа ASP.NET предлагает несколько приемов управления состоянием, которые продолжают работать в точности так же и внутри ASP.NET MVC. Для управления состоянием в приложениях ASP.NET MVC вы вольны использовать кеш и состояние сеанса ASP.NET или даже API-интерфейсы `HttpContext.Items`.

Развертывание и исполняющая среда

Веб-сайты ASP.NET MVC развертываются в той же самой разновидности производственной среды, что и приложения Web Forms. Это означает, что все ваши ранее полученные знания о развертывании и сопровождении приложений ASP.NET (т.е. использование Internet Information Server (IIS), пул приложений .NET, трассировка, поиск и устранение неполадок и даже папки `bin` сборок) по-прежнему актуальны при развертывании и сопровождении приложений ASP.NET MVC. Хотя приложения ASP.NET MVC и Web Forms соответствуют довольно разным архитектурам, в конце концов, все они представляют собой код .NET, который развертывается и выполняется для обработки запросов HTTP.

Больше различий, чем сходства

Количество сходных черт, указанных в предыдущем разделе, могло натолкнуть на мысль о том, что ASP.NET MVC и Web Forms являются практически идентичными инфраструктурами. Однако если проанализировать обе эти инфраструктуры более глубоко, становится вполне очевидным, что они намного больше отличаются, нежели подобны. Данный факт отражен в табл. 2.1, где сравниваются основные компоненты инфраструктур.

Таблица 2.1. Фундаментальные отличия между ASP.NET MVC и Web Forms

Web Forms	ASP.NET MVC
Представления тесно связаны с логикой	Представления и логика сохраняются разделенными
Страницы (URL на основе файлов)	Контроллеры (URL на основе маршрутов)
Управление состоянием (View State)	Отсутствие автоматического управления состоянием
Синтаксис Web Forms	Настраиваемый синтаксис (по умолчанию Razor)
Серверные элементы управления	Вспомогательные методы HTML
Мастер-страницы	Компоновки
Пользовательские элементы управления	Частичные представления



Обратите внимание, что поскольку обе инфраструктуры построены на основе ASP.NET, во многих отношениях вполне возможно сделать поведение Web Forms более похожим на ASP.NET MVC и наоборот. Другими словами, вы, безусловно, можете применять многие приемы MVC к приложениям Web Forms и наоборот. Однако имейте в виду, что сравнения, проведенные в этой главе, относятся к стандартной практике разработки для каждой инфраструктуры, т.е. к тому, как это демонстрируется в “официальных” каналах, таких как документация и учебники по API-интерфейсам.

Разделение логики приложения и логики представления

Наиболее важные отличия между ASP.NET MVC и Web Forms касаются используемых ими фундаментальных архитектурных концепций.

Например, инфраструктура Web Forms была введена для обеспечения лучшего разделения ответственности, чем предшествующая технология ASP Framework, которая, в сущности, вынуждала разработчиков комбинировать бизнес-логику и разметку в одном и том же файле. Инфраструктура Web Forms не только предоставила возможность отделения бизнес-логики от разметки, но также предложила намного более мощную платформу разработки .NET Framework, позволяющую писать код, который управляет этой бизнес-логикой. Однако, несмотря на все преимущества Web Forms по сравнению с традиционными сценариями ASP, архитектура Web Forms по-прежнему очень сильно связана на “страницы”; и хотя код перемещен в другое местоположение, все еще довольно трудно действительно отделить бизнес-логику, обрабатывающую запрос, от представления, которое видит пользователя.

С другой стороны, инфраструктура ASP.NET MVC с самого начала построена с учетом концепции разделения ответственности, поддерживаемой слабо связанными компонентами, которые взаимодействуют друг с другом для обработки запроса. Такой подход не только приносит пользу жизненному циклу разработки, расширяя возможности по созданию индивидуальных компонентов в относительной изоляции и последующему их тестированию, но также позволяет обрабатывать запросы динамическим образом, как будет показано в следующем разделе.

URL и маршрутизация

В рамках архитектуры Web Forms каждый внешний URL веб-сайта представлен физической страницей .aspx, и каждая из этих страниц тесно связана с одиночным необязательным классом (называемым “отделенным кодом”), который содержит логику для этой страницы. Страницы не могут динамически выбирать класс, к которому они привязаны, и классы отделенного кода не могут визуализировать другие представления.

Подход на основе страниц Web Forms является полной противоположностью способу обработки запросов в ASP.NET MVC, который полагается на потенциально сложные правила маршрутизации для динамического отображения внешних URL на соответствующие действия контроллеров и предоставления логике действий контроллеров возможности определять, какие представления будут отображаться пользователю.

Например, конвейер может реагировать по-разному на переход пользователя к URL вида /auctions/details/123 в его браузере и на запрос AJAX того же самого URL, просто основываясь на том факте, что запрос AJAX содержит специальный заголовок, который идентифицирует его как таковой. В случае если переход непосредственно на указанный URL осуществляется пользователь, сервер может возвратить полную веб-страницу с разметкой HTML, кодом JavaScript и стилями CSS, тогда как на запрос AJAX сервер может ответить предоставлением только сериализованных данных для запрошенного объекта Auction. Более того, сервер может даже сериализовать объект Auction в разных форматах (например, JSON или XML), основываясь на таких аспектах запроса, как значение параметра строки запроса.

Такой вид динамической обработки весьма непросто поддерживать в Web Forms Framework, не пользуясь модулями или обработчиками HTTP.



Функциональность маршрутизации ASP.NET не ограничивается приложениями ASP.NET MVC – ее также можно применять для настройки URL в приложениях Web Forms.

Главное отличие между этими двумя применениями заключается в том, что в архитектуре ASP.NET MVC маршрутизация является настолько важным компонентом, что инфраструктура просто не сможет функционировать без нее. С другой стороны, приложения Web Forms в основном используют маршрутизацию для устранения ограничений, связанных с путями файлов страниц, и достижения лучшего контроля над URL.

Управление состоянием

Пожалуй, наиболее спорным отличием между Web Forms и ASP.NET MVC является способ поддержки пользовательского состояния между запросами – в частности, то, что в ASP.NET MVC отброшено состояние представления. Как можно было удалить нечто столь фундаментальное? Если кратко, то причина в том, что ASP.NET MVC принимает не поддерживающую состояния природу веб. Чтобы лучше понять сказанное, давайте вернемся еще раз к истории появления Web Forms.

В дополнение к предоставлению лучшей, чем ASP, платформы разработки, одна из первоначальных целей Web Forms Framework заключалась в том, чтобы перенести в веб технологии разработки приложений “толстого” клиента – в частности, перетаскивание и другие концепции быстрой разработки приложений (Rapid Application Development – RAD).

Чтобы сблизить веб и приемы разработки собственных клиентских приложений, в Web Forms был создан уровень абстракции поверх фундаментальных концепций веб-разработки, таких как HTML-разметка и оформление посредством стилей. Одна из наиболее значимых концепций разработки собственных клиентских приложений состоит в том, что собственное приложение поддерживает состояние, т.е. приложение осведомлено о состоянии взаимодействия с пользователем и может даже сохранять это состояние с целью повторного использования в других своих экземплярах.

С другой стороны, веб-сеть основана на запросах HTTP, при этом каждый запрос включает одиночный запрос клиента и одиночный ответ сервера. Веб-сервер должен обрабатывать каждый запрос HTTP изолированно, не зная ни предыдущие, ни последующие сообщения, которыми будет обмениваться заданный клиент; это предотвращает необходимость во взаимодействии сервера и клиента с целью выяснения предыдущих таких взаимодействий.

Для того чтобы использовать взаимодействия, поддерживающие состояние, в среде, которая состояние не поддерживает, должна быть применена абстракция – именно поэтому появилось состояние представления (View State). Попросту говоря, состояние представления сериализирует состояние взаимодействия между клиентом и сервером и сохраняет эту информацию в скрытом поле формы на каждой странице, которую сервер отправляет клиенту. Затем на клиента возлагается ответственность за передачу этого сериализованного состояния с каждым последующим запросом при взаимодействии.

Вместо того чтобы дублировать эту абстракцию, ASP.NET MVC принимает не поддерживающую состояние природу веб и не предоставляет готовых альтернатив состоянию представления кроме таких технологий серверной стороны, как кеширование и состояние сеанса. Взамен ASP.NET MVC ожидает, что запросы включают все данные, которые нужны серверу для их обработки. Например, вместо извлечения объекта Auction из базы данных и его сериализации для клиента, чтобы тот затем отправ-

лял объект Auction в последующих запросах, ответы ASP.NET MVC могут содержать просто идентификатор Auction, который будет применяться контроллером ASP.NET MVC, обрабатывающим запрос, для извлечения объекта Auction из базы данных в течение каждого запроса.

Очевидно, с обоими подходами связаны свои преимущества и компромиссы. Хотя состояние представления существенно упрощает многие взаимодействия для клиентов, содержащиеся внутри состояния данные могут довольно быстро стать громоздкими, приводя к трате значительной части полосы пропускания на данные, которые могут никогда и не использоваться. Другими словами, состояние представления упрощает жизнь разработчиков, не заставляя думать о поддержке состояния, однако делает это за счет увеличения трафика, особенно если разработчики особо не интересуются, какие данные страниц сохраняются.

С другой стороны, подход ASP.NET MVC может уменьшить размеры страниц, но за счет увеличения объема обработки на сервере и (возможно) количества запросов к базе данных, предназначенных для полного восстановления состояния запроса на сервере.

Визуализация HTML-разметки

Каждый веб-сайт отличается от других, но всем веб-сайтам присуща одна общая черта – они должны генерировать HTML-разметку. Одним из главных требований к любой инфраструктуре веб-приложений является обеспечение для разработчиков возможности максимально эффективной визуализации HTML-разметки. И ASP.NET MVC, и Web Forms делают это очень хорошо, но совершенно разными путями.

Представления Web Forms легко отличить с первого же взгляда; почти все компоненты представлений Web Forms – от дескрипторов `<label>` до частично визуализированных разделов AJAX – используют для визуализации HTML-разметки серверные дескрипторы.

Например, ниже приведен пример HTML-представления типичной страницы Web Forms:

```
<%@ Page Title="EBuy Auction Listings" Language="C#" AutoEventWireup="true"
MasterPageFile="~/Layout.master"
CodeBehind="Default.aspx.cs" Inherits="EBuy.Website._Default" %>
<%@ Register TagPrefix="uc" TagName="SecondaryActions"
Src("~/Controls/SecondaryActions.ascx") %>

<asp:Content ID="HeaderContent" runat="server" ContentPlaceHolderID="HeadContent">
    <uc:SecondaryActions runat="server" />
</asp:Content>

<asp:Content ID="BodyContent" runat="server" ContentPlaceHolderID="MainContent">
    <div class="container">
        <header>
            <h3>Auctions</h3>
        </header>
        <asp:Repeater id="auctions" runat="server" DataSource="<%# Auctions %>">
            <HeaderTemplate>
                <ul id="auctions">
            </HeaderTemplate>
            <ItemTemplate>
                <li>
```

```

<h4 class="title">
<asp:HyperLink runat="server"
    NavigateUrl='<%# DataBinder.Eval(Container.DataItem, "Url") %>'>
    <%# Container.DataItem("title") %>
</asp:HyperLink>
</h4>
</li>
</ItemTemplate>
<FooterTemplate>
    </ul>
</FooterTemplate>
</asp:Repeater>
</div>
<script type="text/javascript"
    src="<%: RelativeUrl("~/scripts/jquery.js") %>"></script>
</asp:Content>

```

А теперь сравните это представление с его эквивалентом Razor:

```

@model IEnumerable<Auction>
 @{
    ViewBag.Title = "EBuy Auction Listings";
}
@section HeaderContent {
    @Html.Partial("SecondaryActions")
}
<div class="container">
    <header>
        <h3>Auctions</h3>
    </header>
    <ul id="auctions">
        @foreach(var auction in Model.Auctions) {
            <li>
                <h4 class="title">
                    <a href="@auction.Url">@auction.Title</a>
                </h4>
            </li>
        }
    </ul>
</div>
<script type="text/javascript" src "~/scripts/jquery.js"></script>

```

Оба примера визуализируют одинаковую HTML-разметку, хотя делают это совершенно разными способами.

Обратите внимание, что для генерации HTML-разметки в Razor применяется подход, намного более ориентированный на код, который опирается на кодовые конструкции вроде цикла `foreach`, а не на специальные HTML-дескрипторы серверной стороны, такие как элемент управления `<asp:Repeater>`.

Например, давайте сравним, как Web Forms и Razor визуализируют простой анкерный дескриптор для произвольного URL. Вот как это делается в Web Forms:

```

<asp:HyperLink runat="server" NavigateUrl='<%# auction.Url %>'>
    <%: auction.Title %>
</asp:HyperLink>

```

А вот версия Razor:

```
<a href="@auction.Url">@auction.Title</a>
```

Эти два примера наглядно демонстрируют отличия в подходах, принятых двумя инфраструктурами: Razor помогает разработчикам записывать саму HTML-разметку, тогда как представления Web Forms зависят от декларативной разметки серверных элементов управления для визуализации их HTML-разметки.

Сравнение вспомогательных методов HTML и серверных элементов управления

Давайте ненадолго остановимся на последнем утверждении, т.к. оно очень важно. Благодаря абстракции Web Forms, целые веб-страницы можно писать вообще без HTML-разметки, но с использованием полного набора *серверных элементов управления* – компонентов, визуализирующих HTML-разметку, которые доступны в форме декларативной разметки (подобной показанному ранее дескриптору `<asp:Hyperlink>`) и предоставляются инфраструктурой в готовом виде для решения большинства потребностей, связанных с HTML.

При этом подход ASP.NET MVC к визуализации разметки особенно отличается от подхода Web Forms, поскольку он ожидает, что крупные порции HTML-разметки, которая отправляется в браузер, будут записываться разработчиками. Однако это не означает, что ASP.NET MVC не может генерировать некоторую HTML-разметку подобного рода. Для таких случаев предусмотрены *вспомогательные методы HTML* – логика визуализации HTML-разметки, доступная в виде *расширяющих методов*, которые разработчики могут вызывать в своих представлениях для генерации HTML-разметки там, где она необходима.

Логически вспомогательные методы HTML и серверные элементы управления почти идентичны: они являются кодовыми компонентами, которые выполняются внутри представления для генерации HTML-разметки, поэтому разработчикам не приходится писать ее вручную.

Главное отличие между вспомогательными методами HTML и серверными элементами управления связано с их технической реализацией. В то время как серверные элементы управления являются полноценными классами, производными от определенного базового класса, вспомогательные методы HTML доступны в форме расширяющих методов объекта `HtmlHelper`, присутствующего в представлениях ASP.NET MVC.

Еще одно важное отличие между вспомогательными методами HTML и серверными элементами управления касается того, о чем мы неоднократно упоминали в этой главе: большинство серверных элементов управления используют состояние представления в том или ином виде, а вспомогательные методы HTML должны функционировать без него.

За рамками этих отличий наиболее распространенные серверные элементы управления имеют эквивалентные вспомогательные методы HTML.

Например, в качестве противоположности дескриптору `<asp:HyperLink>` из Web Forms инфраструктура ASP.NET MVC предлагает метод `Html.ActionLink()`:

```
@Html.ActionLink(auction.Title, "Details", "Auction")
```

В этом примере методу передаются три параметра: текст для отображения пользователю, имя действия (`Details`) и имя контроллера (`Auction`), которые должны применяться для построения URL. Результат визуализации выглядит следующим образом:

```
<a href="/Auction/Details/">My Auction Title</a>
```

В отличие от Web Forms, в ASP.NET MVC не предусмотрены вспомогательные методы HTML для генерации абсолютно всей разметки, но те методы, которые доступны, позволяют решить большинство нетривиальных задач генерации разметки.



Аналогично серверным элементам управления Web Forms, можно создавать собственные вспомогательные методы HTML для инкапсуляции частично используемой логики визуализации.

Сравнение частичных представлений и пользовательских элементов управления

Точно так же, как вспомогательные методы HTML эквивалентны серверным элементам управления Web Forms, *частичные представления* ASP.NET MVC и пользовательские элементы управления Web Forms создают, в сущности, одно и то же: разделы представления, которые хранятся в отдельных файлах, позволяя разработчикам разбивать одно представление на множество небольших представлений и затем собирать их вместе во время выполнения. Подобно пользовательским элементам управления, частичные представления предлагают удобный способ инкапсуляции порций представления и даже повторного использования этих разделов во множестве представлений.

Сравнение компоновок и мастер-страниц

Наконец, мы подошли к одной из самых важных и фундаментальных концепций, связанных с представлениями: возможности определять структуру, компоновку и общую тему сайта и разделять это определение между всеми страницами на сайте. Да, имеются в виду *мастер-страницы* или, как их называют в мире ASP.NET MVC, *компоненты*.

Компоновки ASP.NET MVC позволяют разработчикам указывать фрагмент HTML-разметки, который будет располагаться на любой странице. Подобно мастер-страницам, компоновки ASP.NET MVC дают возможность разработчикам устанавливать контент во множестве разделов страницы. И, подобно страницам контента Web Forms, представления ASP.NET MVC обычно указывают, внутри какой компоновки они ожидают быть визуализированными, но только с одним большим отличием: это указание выступает лишь в качестве “рекомендации” для конвейера ASP.NET MVC, который волен изменить компоновку представления по собственному усмотрению — включая даже ее полное удаление с целью визуализации представления вообще без компоновки (как в случае запроса AJAX).

Реализация представлений ASP.NET MVC с использованием синтаксиса Web Forms

Теперь, когда вы добрались до конца этой главы, самое время выдать вам один секрет: Razor не является единственным способом реализации представлений ASP.NET MVC. На самом деле, если вы пока еще не готовы полностью отказаться от Web Forms, инфраструктура ASP.NET MVC предоставляет возможность продолжить пользоваться синтаксисом Web Forms при разработке представлений ASP.NET MVC.

Прежде всего, имейте в виду, что вы будете применять только *синтаксис Web Forms*, но не *инфраструктуру Web Forms*. Другими словами, все, что было сказано в этой главе, остается верным независимо от используемого синтаксиса: маршруты URL по-прежнему выбирают контроллеры для запуска, которые, в свою очередь, определяют

представление для отображения; серверные элементы управления ASP.NET не будут работать должным образом и, что важнее всего, состояние представления останется незаполненным.

Написание представлений ASP.NET MVC с помощью синтаксиса Web Forms означает использование синтаксиса кода <% %> (вместо синтаксиса @, принятого в Razor) внутри файлов .aspx, .ascx и даже .master. Однако мастер-страницы являются исключением из этого правила — они продолжают работать в основном точно так же, как в приложениях Web Forms, поэтому представления ASP.NET MVC по-прежнему могут пользоваться преимуществами отделения разметки уровня сайта от разметки, сгенерированной индивидуальными страницами контента.



Инфраструктура ASP.NET MVC не только поддерживает представления в синтаксисе Razor и Web Forms, но их можно даже смешивать! Например, какое-то представление может вызывать @Html.Partial("MyPartial"), ссылаясь на представление *MyPartial*, которое является пользовательским элементом управления Web Forms. Тем не менее, стили компоновок смешивать нельзя: подходы, положенные в основу компоновок Razor и мастер-страниц Web Forms, не совместимы друг с другом; это означает, что представления Razor не могут ссылаться на мастер-страницы Web Forms, а страницы контента Web Forms не могут ссылаться на компоновки Razor.

Несколько предостережений

Хотя в большей части этой главы обсуждалось, насколько текущие знания Web Forms могут помочь в работе с ASP.NET MVC, следует принимать во внимание и отрицательную сторону многолетнего опыта применения Web Forms: при всей общности и подобии, архитектура и цели, заложенные в инфраструктуры ASP.NET MVC и Web Forms, фундаментально отличаются. Если вы привыкли к разработке “методом Web Forms” и попытаетесь применить эти же подходы в ASP.NET MVC, то возникнут проблемы.

Как было указано выше, самая важная и “опасная” разница между этими инфраструктурами заключается в том, что Web Forms стремится ввести и поддерживать состояние, в то время как ASP.NET MVC этого не делает. С технической точки зрения это все сводится к тому, что при переходе от Web Forms на ASP.NET MVC больше нет состояния представления, и большая часть поддержки состояния, обеспечиваемая Web Forms, работать не будет.

Более того, вы должны помнить о том, что именно было помещено в состояние представления. Зачастую удобство состояния представления превращает его в свалку для данных, которые будут повторно использоваться в запросах. Если так и произошло, понадобится вывести “способ ASP.NET MVC” для решения той же задачи и найти другие места для временного хранения этих данных, такие как *состояние сеанса* или *кеш приложения*.

Следующее крупное отличие касается способа написания разметки представления. В то время как Web Forms основывается на дескрипторах для серверных и пользовательских элементов управления, в ASP.NET MVC применяются вспомогательные методы HTML и частичные представления. Несмотря на концептуальное подобие, эти два подхода не являются взаимозаменяемыми и они не должны (и часто не могут) смешиваться друг с другом.

Использование синтаксиса Razor помогает избежать этой проблемы, делая совершенно очевидным тот факт, что вы не пишете страницы Web Forms. Однако ASP.NET

предлагает *механизм представлений ASPX*, который позволяет применять синтаксис Web Forms для создания представлений ASP.NET MVC. Хотя механизм представлений ASPX выполняет всю работу по визуализации HTML-разметки, если вы не проявите должной внимательности, то можете обнаружить, что к своему огорчению используете части Web Forms Framework. Во избежание путаницы из-за того, что какая-то разметка может выглядеть как Web Forms, во всех представлениях, показанных в этой книге, будет применяться синтаксис Razor.

Несмотря на свою важность, упомянутые выше проблемы не должны удерживать вас от изучения и использования ASP.NET MVC или даже от применения механизма представлений ASPX, если синтаксис Web Forms кажется вам более удобным. Просто при реализации функциональности ASP.NET MVC имейте в виду концепции, изложенные в этой главе. Продолжая постоянно задавать себе вопросы вроде “Вписывается ли это в подход MVC?” или “В полном ли объеме используются возможности инфраструктуры ASP.NET MVC?”, вы должны в полной мере задействовать свои знания Web Forms и в то же время избежать указанных выше проблем.

Резюме

Поскольку инфраструктуры ASP.NET MVC и Web Forms разделяют общую базу, разработчики Web Forms имеют солидную опору при изучении ASP.NET MVC. В этой главе были показаны сходные черты этих двух инфраструктур, в также демонстрировались их отличия в обработке определенных сценариев. В приложении А объясняется, как извлечь выгоду из разделяемой инфраструктурой основной функциональности на примерах простого переноса существующих приложений Web Forms в ASP.NET MVC.

Работа с данными

Редко когда удается встретить приложение, которое бы каким-либо образом не имело дело с данными, поэтому не должен вызывать удивление тот факт, что ASP.NET MVC предоставляет великолепную поддержку работы с данными на всех уровнях инфраструктуры. В этой главе мы рассмотрим инструменты, обеспечивающие такую поддержку, и покажем, как использовать их в сценариях, управляемых данными, за счет добавления этой функциональности к образцовому приложению EBiu.

Поскольку EBiu является сайтом онлайновых аукционов, наиболее важным его сценарием является предоставление пользователям возможности создавать списки аукционных товаров, содержащие детальные сведения по каждому элементу, который они желают продать. Итак, давайте посмотрим, как ASP.NET MVC может помочь в поддержке этого важного сценария.

Построение формы

Концепция HTML-формы столь же стара, как сама веб-сеть. Хотя в настоящее время браузеры стали более функциональными, а HTML-форму можно стилизовать и снабдить поведением с помощью JavaScript таким способом, который еще пять лет назад казался невозможным, в ее основе по-прежнему лежит набор старых добрых полей, готовых к заполнению и обратной отправке серверу.

Несмотря на то что ASP.NET MVC приветствует написание большей части HTML-разметки “вручную”, эта инфраструктура предлагает набор вспомогательных методов HTML, позволяющих генерировать разметку для HTML-форм, среди которых `Html.TextBox`, `Html.Password` и `Html.HiddenField`. Кроме того, в ASP.NET MVC имеется несколько “более интеллектуальных” вспомогательных методов, таких как `Html.LabelFor` и `Html.EditorFor`, которые динамически определяют подходящую HTML-разметку на основе имени и типа переданного свойства модели.

Именно эти вспомогательные методы будут использоваться в примере веб-сайта EBiu для построения HTML-формы, которая позволит пользователям выполнять отправку в адрес действия `AuctionsController.Create` для создания новых аукционных товаров. Чтобы увидеть данные вспомогательные методы в действии, добавьте новое представление по имени `Create.cshtml` и заполните его следующей разметкой:

```
<h2>Create Auction</h2>
@using (Html.BeginForm() ) {
    <p>
        @Html.LabelFor(model => model.Title)
        @Html.EditorFor(model => model.Title)
    </p>
```

```

<p>
    @Html.LabelFor(model => model.Description)
    @Html.EditorFor(model => model.Description)
</p>
<p>
    @Html.LabelFor(model => model.StartPrice)
    @Html.EditorFor(model => model.StartPrice)
</p>
<p>
    @Html.LabelFor(model => model.EndTime)
    @Html.EditorFor(model => model.EndTime)
</p>
<p>
    <input type="submit" value="Create" />
</p>
}

```

Затем добавьте в контроллер приведенное ниже действие для визуализации этого представления:

```

[HttpGet]
public ActionResult Create()
{
    return View();
}

```

Показанное выше представление визуализируется в следующую HTML-разметку, предназначенную для браузера:

```

<h2>Create Auction</h2>
<form action="/auction/create" method="post">
    <p>
        <label for="Title">Title</label>
        <input id="Title" name="Title" type="text" value="">
    </p>
    <p>
        <label for="Description">Description</label>
        <input id="Description" name="Description" type="text" value="">
    </p>
    <p>
        <label for="StartPrice">StartPrice</label>
        <input id="StartPrice" name="StartPrice" type="text" value="">
    </p>
    <p>
        <label for="EndTime">EndTime</label>
        <input id="EndTime" name="EndTime" type="text" value="">
    </p>
    <p>
        <input type="submit" value="Create">
    </p>
</form>

```

Пользователь затем заполняет эту форму значениями и отправляет ее действию `/auctions/create`. Хотя с точки зрения браузера это выглядит как отправка формы самой себе (URL для визуализации формы изначально установлен также в `/auctions/create`), именно здесь в игру вступает второе действие контроллера `Create` с атрибутом `HttpPostAttribute`, сообщающее ASP.NET MVC о том, что это перегруженная версия, которая обрабатывает действие POST отправки формы.

А теперь самое время действительно *сделать* что-нибудь со значениями отправленной формы — но что?

Обработка отправок формы

Перед тем как можно будет работать со значениями, отправленными контроллеру, понадобится извлечь их из запроса. Как было показано в разделе “Параметры действий” главы 1, простейший способ предусматривает использование модели в качестве параметра действия и, к счастью, модель уже создана: вспомните класс `Auction` из раздела “Модели” в главе 1.

Для привязки к созданному ранее классу `Auction` просто укажите параметр типа `Auction` в числе параметров действия контроллера `Create`:

```
[HttpPost]
public ActionResult Create(Auction auction)
{
    // Создать объект Auction в базе данных.
    return View(auction);
}
```

Наиболее важным аспектом модели `Auction` в этой точке является тот факт, что имена свойств (`Title`, `Description` и т.д.) соответствуют именам полей формы, которые были отправлены действию `Create`. Имена свойств критически важны, поскольку привязка модели ASP.NET MVC пытается заполнить их значения из полей формы с совпадающими именами.

Если вы запустите это приложение, выполните отправку почты и обновите страницу, то увидите, что модель `Auction` заполнилась введенными ранее значениями. В этот момент действие просто возвращает заполненный параметр `auction` обратно представлению, которое может применяться для отображения значений формы пользователю с целью подтверждения отправки.

Это приближает нас на один шаг к получению приложения, которое делает что-то полезное, однако по-прежнему остается многое чего нужно реализовать, начиная с действительного *сохранения* данных.

Сохранение данных в базе

Хотя инфраструктура ASP.NET MVC Framework не имеет никаких встроенных средств доступа к данным, существует множество популярных библиотек .NET доступа к данным, которые помогут упростить работу с базой данных.

Одной из таких библиотек является *Entity Framework* (EF) от Microsoft. Библиотека Entity Framework — это простая и гибкая инфраструктура *объектно-реляционного отображения* (object relational mapping — ORM), которая позволяет разработчикам запрашивать и обновлять данные в базе объектно-ориентированным путем. Более того, Entity Framework в действительности представляет собой часть платформы .NET Framework, с полной поддержкой и обилием доступной документации, обеспечивающей Microsoft.

Инфраструктура Entity Framework предлагает несколько разных подходов к определению модели данных и применению этой модели для доступа в базу данных, но, пожалуй, самым интригующим подходом является *Code First* (“сначала код”). Образ мышления, заложенный в разработку *Code First*, заключается в том, что модель приложения является центральной частью и движущей силой всего происходящего во время разработки.

Entity Framework Code First: соглашение по конфигурации

При разработке Code First взаимодействие осуществляется через простые классы моделей (также называемые традиционными объектами CLR (Plain Old CLR Object – POCO)). Подход Code First в Entity Framework заходит настолько далеко, что даже генерирует на основе модели схему базы данных и использует эту схему для создания базы данных и ее сущностей (таблиц, отношений и т.д.) при запуске приложения.

Подход Code First делает это за счет следования определенным соглашениям, которые автоматически оценивают различные свойства и классы, образующие уровень моделей, с целью выяснения, каким образом информация в этих моделях должна быть сохранена, и даже как отношения между разными классами моделей могут быть эффективно представлены в терминах отношений базы данных.

Например, в образцовом приложении EBuy класс Auction отображается на таблицу базы данных Auctions и все его свойства представляют столбцы в этой таблице. Имена таблицы и столбцов автоматически выводятся из имен класса и его членов.

Показанная ранее модель Auction очень проста, но с ростом потребностей приложения сложность модели также будет возрастать: мы добавим дополнительные свойства, бизнес-логику и даже отношения с другими моделями. Однако это не проблема для Entity Framework Code First, поскольку это средство обычно способно обрабатывать более сложные модели с той же легкостью, что и простые. В главе 8 в простую модель Auction, показанную в этой главе, привносится более реалистичная сложность и демонстрируется, что подход Entity Framework Code First обладает возможностью обработки этих более сложных отображений (а также объясняется, что делать, если он не справляется с ними).

Создание уровня доступа к данным с помощью Entity Framework Code First

В основе подхода Entity Framework Code First лежит класс System.Data.Entity.DbContext. Этот класс (или созданные производные от него классы) выступает в качестве шлюза к базе данных, предоставляя все необходимые действия, связанные с данными. Чтобы приступить к использованию класса DbContext, понадобится создать собственный класс, производный от него, который на самом деле довольно прост:

```
using System.Data.Entity;
public class EbuyDataContext : DbContext
{
    public DbSet<Auction> Auctions { get; set; }
}
```

В этом примере (EbuyDataContext.cs) мы создали специальный класс контекста данных по имени EbuyDataContext, производный от DbContext. Этот отдельный класс определяет свойство System.Data.Entity.DbSet<T>, где T – это сущность, которая будет редактироваться и сохраняться в базе данных. В предшествующем примере мы определили System.Data.Entity.DbSet<Auction> для указания на то, что приложение нуждается в сохранении и редактировании экземпляров класса Auction в базе данных. Однако в контексте данных можно определять более одной сущности, и по мере продвижения разработки мы будем добавлять в класс EbuyDataContext дополнительные сущности (или свойства DbSet).

Если создание специального контекста данных осуществляется легко, то его использование еще легче, как продемонстрировано в следующем примере. В следующем фрагменте действие контроллера Create модифицируется для сохранения отправлен-

ного объекта Auction в базе данных, для чего объект Auction просто добавляется в коллекцию EbuyDataContext.Auctions с последующим сохранением изменений:

```
[HttpPost]
public ActionResult Create(Auction auction)
{
    var db = new EbuyDataContext();
    db.Auctions.Add(auction);
    db.SaveChanges();
    return View(auction);
}
```

На этот раз после запуска приложения и отправки заполненной формы в таблице Auctions базы данных появится новая строка, содержащая информацию, отправленную в форме.

Если вы продолжите пользоваться этим примером и поэкспериментируете с различными значениями в полях формы, то заметите, что привязка моделей ASP.NET MVC является весьма терпимой к ошибкам, позволяя вводить все что угодно и молча отказывая, когда она не может преобразовать отправленные значения формы в строгие типы (например, в ситуации, когда пользователь вводит строку ABC в поле типа int). Если необходим более строгий контроль над тем, какие данные сохраняются в базе, потребуется применять проверку достоверности данных к модели.

Проверка достоверности данных

Что касается данных, то обычно существует ряд применяемых правил и ограничений, таких как поля, которые не должны быть пустыми или значения которых должны находиться в заданном диапазоне, чтобы рассматриваться как “допустимые”. Естественно, ASP.NET MVC распознает такие важные концепции, интегрируя их прямо в процесс обработки каждого запроса.

В качестве части процесса выполнения действия контроллера инфраструктура ASP.NET MVC Framework проверяет достоверность данных, которые передаются этому действию контроллера, заполняя объект ModelState любыми обнаруженными ошибками и передавая этот объект контроллеру. Затем действия контроллера могут запросить ModelState для выяснения допустимости запроса и отреагировать соответствующим образом, например, сохранить допустимый объект в базе данных или вернуть пользователя в исходную форму для исправления ошибок проверки достоверности, зафиксированных в недопустимом запросе.

Ниже приведен пример действия AuctionsController.Create, обновленного для проверки словаря ModelState с применением только что описанной логики “сохранить или исправить”:

```
[HttpPost]
public ActionResult Create(Auction auction)
{
    if (ModelState.IsValid)
    {
        var db = new EbuyDataContext();
        db.Auctions.Add(auction);
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    return View(auction);
}
```

Инфраструктура ASP.NET MVC Framework – это не единственное средство, которое может добавлять ошибки проверки достоверности в ModelState. Разработчики могут запускать собственную логику для обнаружения проблем, которые инфраструктура не может перехватить, и вручную добавлять информацию об ошибках, используя такой метод:

```
ModelState.AddModelError(string key, string message)
```

Пусть, например, существует требование, что аукционы должны длиться, по крайней мере, один день. Другими словами, значение свойства EndTime объекта Auction должно превышать текущее время плюс один день.

Действие AuctionsController.Create может явно проверять это перед тем, как пытаться сохранить объект Auction, и предусматривать специальное сообщение об ошибке, когда такая ситуация возникает:

```
[HttpPost]
public ActionResult Create(Auction auction)
{
    if (auction.EndTime <= DateTime.Now.AddDays(1))
    {
        ModelState.AddModelError(
            "EndTime",
            "Auction must be at least one day long"
        );
    }

    if (ModelState.IsValid)
    {
        var db = new EbuyDataContext();
        db.Auctions.Add(auction);
        db.SaveChanges();
        return RedirectToAction("Index");
    }

    return View(auction);
}
```

Хотя этот подход работает довольно неплохо, он приводит к нарушению принципа разделения ответственности в приложении. В частности, контроллеры не должны содержать бизнес-логику подобного рода: бизнес-логика относится к модели. Итак, давайте перенесем бизнес-логику в модель.

Указание бизнес-правил с помощью аннотаций данных

Практика обеспечения качества данных – проверка достоверности данных – является настолько распространенной задачей при разработке приложений, что для разработчиков вполне естественно обращаться к одной из многих доступных инфраструктур для получения помощи в определении и выполнении логики проверки достоверности данных наиболее эффективным образом.

На самом деле это настолько общая потребность, что в рамках ядра .NET Framework поставляется очень эффективный и простой в использовании API-интерфейс проверки достоверности данных под названием Data Annotations (аннотации данных). Как должно быть понятно из названия, API-интерфейс Data Annotations предоставляет набор атрибутов .NET, которые разработчики могут применять к свойствам классов объектов данных. Эти атрибуты предлагают декларативный способ применения правил проверки достоверности непосредственно к модели.

Более того, привязка модели ASP.NET MVC обеспечивает поддержку аннотаций данных без дополнительного конфигурирования. Для демонстрации работы аннотаций данных ASP.NET MVC давайте рассмотрим процесс применения проверки достоверности к классу Auction. Перед началом применения логики проверки достоверности необходимо определить, какие значения ожидаются для свойств класса Auction. Какие поля должны быть обязательными? Имеют ли какие-то поля определенные диапазоны допустимых значений?

Обязательные поля

Поскольку свойства Title и Description класса Auction критически важны для описания продаваемого на аукционе товара, мы применим к этим двум полям аннотацию данных RequiredAttribute, пометив их как поля, которые обязательно должны иметь данные, чтобы считаться допустимыми:

```
[Required]  
public string Title { get; set; }  
  
[Required]  
public string Description { get; set; }
```

В дополнение к пометке поля как обязательного с помощью RequiredAttribute, можно также обеспечить, чтобы строковые значения имели максимальную длину, применив для этого атрибут StringLengthAttribute. Например, было решено, что заголовки аукционных товаров должны сохраняться короткими, не превышая максимальную длину в 50 символов:

```
[Required, StringLength(50)]  
public string Title { get; set; }
```

Если теперь пользователь отправит форму с полем Title, в котором введена строка с более чем 50 символами, средство проверки достоверности модели ASP.NET MVC сообщит об ошибке.

Допустимые диапазоны

Далее рассмотрим стартовую цену аукционного товара: она представляется свойством StartPrice, имеющим тип decimal. Поскольку decimal – это тип значения, свойство StartPrice будет всегда иметь, по крайней мере, значение по умолчанию, равное 0, так что пометка этого свойства как обязательного будет избыточной. Тем не менее, со стартовыми ценами аукционных товаров связана другая логика: эти значения не могут быть отрицательными! Чтобы решить эту проблему, примените атрибут RangeAttribute к полю StartPrice и укажите в качестве минимального значения 1. Так как RangeAttribute требует еще и максимального значения, укажите также верхний предел.

```
[Range(1, 10000)]  
public decimal StartPrice { get; set; }
```

В этом примере используется диапазон типа double, но аннотация RangeAttribute имеет также перегруженную версию (Range (Type type, string min, string max)) для поддержки диапазона любого типа, который реализует интерфейс IComparable и может быть создан путем разбора или преобразования строковых значений. Хорошим примером может служить проверка диапазона дат; например, следующая аннотация гарантирует, что дата находится после определенного момента времени:

```
[Range(typeof(DateTime), "1/1/2012", "12/31/9999")]  
public DateTime EndTime { get; set; }
```

Этот пример обеспечивает, что значение свойства `EndTime` будет, по крайней мере, позже 1 января 2012 г.



Параметрами атрибутов .NET должны быть значения, которые известны на этапе компиляции и не могут вычисляться во время выполнения, что исключает использование таких значений, как `DateTime.Now`, для выяснения, относится ли дата к будущему. Вместо этого мы должны выбрать произвольную дату, например, 1/1/2012, которая хотя и не обеспечит того, что введенная дата относится к моменту, находящемуся после отправки формы, но, по крайней мере, позволит избежать ввода дат из далекого прошлого.

Этот недостаток точности является компромиссом, на который пришлось пойти, чтобы иметь возможность пользоваться `RangeAttribute`. Если сложившаяся ситуация требует большей точности, придется прибегнуть к атрибуту `CustomValidationAttribute`, который позволяет выполнять произвольную логику для проверки достоверности свойств. Хотя возможность выполнения произвольного кода посредством `CustomValidatorAttribute`, несомненно, удобна, она представляет собой менее декларативный подход, который ограничивает информацию, доступную другим компонентам, таким как инфраструктура проверки достоверности на стороне клиента ASP.NET MVC.

Специальные сообщения об ошибках

В заключение важно отметить, что все аннотации данных предоставляют свойство `ErrorMessage`, которое можно использовать для указания сообщения об ошибке, отображаемого пользователю вместо стандартного сообщения об ошибке от API-интерфейса Data Annotations. Укажите желаемое значение для этого свойства в каждой аннотации данных, добавленной к модели.

Финальный класс, включающий все аннотации данных, которые обсуждались в этом разделе, должен выглядеть примерно так:

```
public class Auction
{
    [Required]
    [StringLength(50,
        ErrorMessage = "Title cannot be longer than 50 characters")]
    public string Title { get; set; }

    [Required]
    public string Description { get; set; }

    [Range(1, 10000,
        ErrorMessage = "The auction's starting price must be at least 1")]
    public decimal StartPrice { get; set; }

    public decimal CurrentPrice { get; set; }
    public DateTime EndTime { get; set; }
}
```

Теперь, когда в модели определена вся необходимая логика проверки достоверности, давайте вернемся к контроллеру и представлению и посмотрим, как отображать сообщения об ошибках пользователю.

Отображение сообщений об ошибках проверки достоверности

Вы можете сказать, что добавленные правила проверки достоверности работают, поместив точку останова в действие `Create`, оправив недопустимые значения и проверив свойство `ModelState` на предмет фактического добавления ошибок проверки. Факт возвращения контроллером представления `Create` вместо добавления нового аукционного товара и перенаправления на другую страницу является еще одним доказательством того, что проверочные правила установлены корректно, а инфраструктура проверки достоверности работает. Проблема в том, что хотя представление `Create` может показывать поля с недопустимыми значениями в красной рамке, она все еще не отображает сообщения об ошибках, указывающие конкретные причины их возникновения. Давайте позаботимся об этом.

В качестве напоминания ниже приведена текущая разметка для свойства `Title`:

```
<p>
    @Html.LabelFor(model => model.Title)
    @Html.EditorFor(model => model.Title)
    @ViewData.ModelState["Title"]
</p>
```

Нам необходимо добавить к этой разметке еще одну строку для отображения любых сообщений, связанных с проверкой достоверности свойства `Title`. Простейший способ выяснить, возникли ли ошибки проверки свойства `Title`, заключается в обращении к `ModelState` напрямую — `ViewData.ModelState["Title"]` возвращает объект, содержащий коллекцию ошибок, которые относятся к свойству `Title`.

Затем можно пройти в цикле по этой коллекции, чтобы визуализировать сообщения об ошибках на странице:

```
<p>
    @Html.LabelFor(model => model.Title)
    @Html.EditorFor(model => model.Title)
    @foreach(var error in ViewData.ModelState["Title"].Errors)
    {
        <span class="error">@error.ErrorMessage</span>
    }
</p>
```

Хотя это работает довольно хорошо, ASP.NET MVC предлагает даже лучший подход к визуализации всех ошибок для заданного свойства: вспомогательный метод `Html.ValidationMessage(string modelName)`.

Вспомогательный метод `Html.ValidationMessage()` позволяет заменить весь показанный выше цикл `foreach` единственным вызовом метода и получить тот же самый результат:

```
@Html.ValidationMessageFor(model => model.Title)
```

Добавьте вызов `Html.ValidationMessage()` для каждого свойства в модели. Инфраструктура ASP.NET MVC теперь будет визуализировать все проблемы, возникающие во время проверки достоверности, прямо рядом с полями формы, к которым проверка применяется.

В дополнение к вспомогательному методу `Html.ValidationMessage()` уровня свойств, ASP.NET MVC также предоставляет вспомогательный метод `Html.ValidationSummary()`. Этот метод позволяет визуализировать все ошибки проверки достовер-

ности для формы в одном месте (например, в верхней части формы), предоставляя пользователю сводку по всем проблемам, которые должны быть устраниены, чтобы форма могла быть успешно отправлена.

Вспомогательный метод `Html.ValidationSummary()` очень легко использовать — нужно просто вызвать `Html.ValidationSummary()` там, где должна располагаться сводка:

```
@using (Html.BeginForm())
{
    @Html.ValidationSummary()
    <p>
        @Html.LabelFor(model => model.Title)
        @Html.EditorFor(model => model.Title)
        @Html.ValidationMessageFor(model => model.Title)
    </p>
    <!-- Остальные поля формы... -->
}
```

Если теперь отправить недопустимые значения в полях формы, вы увидите сообщения об ошибках в двух местах (рис. 3.1): в сводке по проверке достоверности (благодаря вызову `Html.ValidationSummary()`) и рядом с полями (благодаря вызову `Html.ValidationMessage()`).

The screenshot shows a 'Create' form with the following details:

- Title:** A text input field with an asterisk (*) indicating it is required.
- Description:** A text input field with an asterisk (*) indicating it is required.
- StartPrice:** A text input field with an error message: "The StartPrice field is required." displayed next to it.
- EndTime:** A text input field.
- Create:** A blue 'Create' button.
- Back to List:** A link labeled 'Back to List'.

Рис. 3.1. Отображение сообщений об ошибках посредством вспомогательного метода `Html.ValidationSummary()`

Если вы хотите избежать отображения дублированных сообщений об ошибках, можете модифицировать вызовы `Html.ValidationMessage()` и указать короткое специальное сообщение, такое как единственный символ звездочки:

```
<p>
    @Html.LabelFor(model => model.Title)
    @Html.EditorFor(model => model.Title)
    @Html.ValidationMessageFor(model => model.Title, "*")
</p>
```

Ниже приведена полная разметка для представления `Create` после добавления всех средств проверки достоверности:

```

<h2>Create Auction</h2>
@using (Html.BeginForm())
{
    @Html.ValidationSummary()
    <p>
        @Html.LabelFor(model => model.Title)
        @Html.EditorFor(model => model.Title)
        @Html.ValidationMessageFor(model => model.Title, "*")
    </p>
    <p>
        @Html.LabelFor(model => model.Description)
        @Html.EditorFor(model => model.Description)
        @Html.ValidationMessageFor(model => model.Description, "*")
    </p>
    <p>
        @Html.LabelFor(model => model.StartPrice)
        @Html.EditorFor(model => model.StartPrice)
        @Html.ValidationMessageFor(model => model.StartPrice)
    </p>
    <p>
        @Html.LabelFor(model => model.EndTime)
        @Html.EditorFor(model => model.EndTime)
        @Html.ValidationMessageFor(model => model.EndTime)
    </p>
    <p>
        <input type="submit" value="Create" />
    </p>
}

```



Вся проверка достоверности, продемонстрированная до сих пор, производилась на стороне сервера, требуя полного цикла обмена между браузером и сервером для обработки каждого потенциально недопустимого запроса и выдачи в качестве ответа полностью визуализированного представления.

Хотя этот подход работает, он определенно не оптимальен. В главе 4 показано, как реализовать проверку достоверности на стороне клиента, чтобы усовершенствовать этот подход и выполнять большинство (если не все) проверок прямо в браузере. Это позволит избежать дополнительных запросов к серверу, сохраняя как полосу пропускания, так и серверные ресурсы.

Резюме

В этой главе речь шла об использовании подхода Entity Framework Code First для создания и обслуживания базы данных приложения. Вы увидели, насколько легко с помощью Entity Framework устанавливать базу данных: это сводится к всего лишь нескольким строкам кода и не требует предварительного рисования диаграммы со схемой или написания SQL-запросов для моделирования и создания базы данных. Было показано, что Entity Framework работает за счет следования соглашениям, а также описаны некоторые базовые соглашения. Также кратко рассматривалось применение средства привязки моделей ASP.NET MVC для автоматического заполнения объектов состояния из входящего запроса.

Разработка на стороне клиента

Сеть Интернет прошла долгий путь, начиная с веб-страниц, которые состояли только из простой HTML-разметки и JavaScript-кода. Популярные веб-приложения, такие как Gmail и Facebook, трансформировали пользовательские ожидания от веб-сайтов: посетителей больше не устраивает один лишь базовый текст, а взамен они требуют развитый интерактивный интерфейс, способный соперничать с интерфейсом, предоставляемым собственными настольными приложениями. По мере роста пользовательских требований, современные браузеры борются за сохранение своих позиций и предпринимают все возможное для внедрения функциональных средств и спецификаций, таких как HTML 5 и CSS3, которые сделали бы такой вид приложений возможным.

Хотя большинство материалов книги сосредоточено на разработке веб-приложений с помощью ASP.NET MVC Framework, в этой главе внимание уделяется фундаментальным основам создания насыщенных веб-приложений на примере использования библиотеки *jQuery* для упрощения разработки на стороне клиента.

Работа с JavaScript

Несовместимости браузеров преследуют разработчиков десятилетиями. Отличия в функциональности и отсутствие стандартов для браузеров привели к появлению многочисленных клиентских библиотек и инфраструктур, которые пытались решить эти проблемы, абстрагируясь от отличий между браузерами, и предоставить по-настоящему стандартный межбраузерный API-интерфейс.

Несомненным фаворитом среди этого множества библиотек является JavaScript-библиотека *jQuery* (<http://jquery.com>), которая, следуя своему лозунгу “Меньше кода, больше работы”, значительно упростила обход модели документного объекта HTML (Document Object Model – DOM), обработку событий, анимацию и взаимодействия AJAX. В версии 3 инфраструктуры ASP.NET MVC Framework библиотека *jQuery* была включена в шаблоны веб-приложений ASP.NET MVC, требуя минимальных усилий для запуска и использования *jQuery*.

Чтобы увидеть, как *jQuery* помогает абстрагироваться от несовместимостей между браузерами, взгляните на следующий код, который пытается выяснить ширину и высоту окна браузера:

```
var innerWidth = window.innerWidth,
    innerHeight = window.innerHeight;
alert("InnerWidth of the window is: " + innerWidth); // ширина окна браузера
alert("InnerHeight of the window is: " + innerHeight); // высота окна браузера
```

Этот сценарий отображает диалоговые окна `alert` с правильными значениями высоты и ширины в большинстве браузеров, однако он генерирует ошибку в версиях Internet Explorer 6–8. Почему?

А причина в том, что указанные версии Internet Explorer (IE) предоставляют ту же самую информацию через свойства `document.documentElement.clientWidth` и `document.documentElement.clientHeight`.

Таким образом, чтобы обеспечить корректную работу этого фрагмента кода во всех браузерах, его потребуется подкорректировать для учета несовместимости IE, как показано в следующем листинге:

```
var innerWidth, innerHeight;
// Все браузеры, исключая IE до версии 9
if (typeof window.innerWidth !== "undefined") {
    innerWidth = window.innerWidth;
    innerHeight = window.innerHeight;
}
else {
    innerWidth = document.documentElement.clientWidth,
    innerHeight = document.documentElement.clientHeight
}
alert("InnerWidth of the window is: " + innerWidth); // ширина окна браузера
alert("InnerHeight of the window is: " + innerHeight); // высота окна браузера
```

Благодаря таким изменениям, сценарий теперь работает правильно во всех основных браузерах. В связи с несоблюдением или разной интерпретацией стандартов W3C, браузеры переполнены особенностями подобного рода. Более старые браузеры хорошо известны несовместимостью или только частичным соответствием со стандартами. И пока новейшие спецификации наподобие HTML 5 и CSS3 находятся в черновом состоянии, современные браузеры спешат предоставить черновые реализации этих спецификаций, используя собственные специфичные для производителя трюки. Представьте себе, если учесть все эти вариации для почти каждого одиночного элемента DOM, к которому осуществляется доступ в приложении. Код не только станет громоздким, но его еще придется постоянно обновлять по мере развития браузеров и стандартов и заполнения пробелов в спецификациях, превращая сопровождение приложения в настоящий кошмар.

Хороший способ изоляции кода приложения от несовместимостей подобного рода предполагает применение инфраструктуры или библиотеки, которая выступает в качестве дополнительного уровня между приложением и кодом для доступа и манипуляций моделью DOM. Библиотека jQuerу является великолепной облегченной инфраструктурой, которая значительно сокращает количество сложностей. Простые API-интерфейсы jQuerу делают доступ и манипулирование моделью DOM простыми и интуитивно понятными и сокращают объем необходимого кода, позволяя сосредоточиться на функциональности приложения, а не заботиться о совместимости браузеров и писать рутинный код.

Ниже показано, как можно переписать предыдущий фрагмент кода с использованием jQuerу:

```
var innerWidth = $(window).width(),
    innerHeight = $(window).height();
alert("InnerWidth of the window is: " + innerWidth); // ширина окна браузера
alert("InnerHeight of the window is: " + innerHeight); // высота окна браузера
```

Этот код довольно похож на чистый код JavaScript со следующими небольшими изменениями:

- объект `window` помещен в специальную функцию `$()`, которая возвращает поддерживающий jQuery объект (эта функция более подробно рассматривается далее в главе);
- в коде используются вызовы функций `.width()` и `.height()` вместо доступа к свойствам `.height` и `.width`.

Теперь вы видите преимущества работы с jQuery – код очень похож на обычный код JavaScript, что упрощает его изучение и понимание, и вдобавок он достаточно мощный, чтобы абстрагироваться от всех межбраузерных проблем. Более того, поскольку обо всех аспектах совместимости браузеров заботится библиотека jQuery, объем кода, требуемого для реализации той же функциональности, сокращается к одной строке на свойство.

Библиотека jQuery упрощает не только *получение* значений свойств, но также их *установку*:

```
// Установить ширину в 480 пикселей.
$(window).width("480px");

// Установить высоту в 940 пикселей.
$(window).height("940px");
```

Обратите внимание, что для установки и получения значений свойств применяются те же самые функции, но с единственной разницей в том, что вызов функции установки принимает параметр с новым значением. Использование одного API-интерфейса для различных способов получения и установки значений делает синтаксис jQuery легким в запоминании и простым в чтении.

Селекторы

Первый шаг в манипулировании элементами DOM заключается в получении ссылки на желаемый элемент. Это можно делать многими путями: через идентификатор элемента, его имя класса или один из его атрибутов, либо за счет использования логики JavaScript для навигации по древовидной структуре DOM с целью нахождения элемента вручную.

Например, ниже показано, как с помощью стандартного кода JavaScript искать элемент DOM по его идентификатору:

```
<div id="myDiv">Hello World!</div>
<script type="text/javascript">
    document.getElementById("myDiv").innerText = "Hello jQuery";
</script>
```

В этом простом примере сначала получается ссылка на элемент `<div>` путем вызова метода `document.getElementById()` с передачей ему идентификатора элемента, и затем изменение внутреннего текста на "Hello jQuery". Этот код будет работать одинаково в каждом браузере, потому что `document.getElementById()` является частью языка JavaScript и поддерживается всеми основными браузерами.

Взгляните на еще один сценарий, где доступ к элементу осуществляется по его имени класса:

```
<div class="normal">Hello World!</div>
<script type="text/javascript">
    document.getElementsByClassName("normal")[0].innerText = "Hello jQuery";
</script>
```

Код выглядит простым — вместо `document.getElementById()` применяется `document.getElementsByClassName()` и производится доступ к первому элементу в массиве для установки свойства `innerText`. Но откуда взялся массив? Метод `document.getElementsByClassName()` возвращает массив, содержащий все элементы с одним и тем же именем класса. К счастью, в приведенном примере имеется только один элемент `<div>`, поэтому мы знаем, что первым элементом этого массива является тот, который мы ищем.

Однако в реальном приложении страницы, скорее всего, будет содержать множество элементов, которые могут как иметь, так и не иметь идентификаторы, и вполне может существовать несколько элементов с одинаковым именем класса (и ряд элементов вообще без имени класса). Элементы будут вложены в контейнерные элементы, подобные `<div>`, `<p>` и ``, в соответствие с проектным решением страницы. Поскольку DOM — это всего лишь иерархическая древовидная структура, в конечном итоге элементы будут вложенными в какой-то контейнер, а все элементы окажутся вложенными в корневой элемент `document`.

Рассмотрим следующий пример:

```
<div class="normal">
    <p>Hello World!</p>
    <div>
        <span>Welcome!</span>
    </div>
</div>
```

Для доступа к элементу `` и изменению его контента понадобится захватить самый внешний элемент `<div>` (имеющий `class="normal"`), пройти по его дочерним узлам, проверить каждый узел, не является ли он ``, и затем провести определенные манипуляции с этим элементом ``.

Типичный код JavaScript для захвата `` выглядит так, как показано ниже:

```
var divNode = document.getElementsByClassName("normal")[0];
for(i=0; i < divNode.childNodes.length; i++) {
    var childDivs = divNode.childNodes[i].getElementsByTagName("div");
    for(j=0; j < childDivs.childNodes.length; j++) {
        var span = childDivs.childNodes[j].getFirstChild();
        return span;
    }
}
```

Весь этот код предназначен для захвата всего лишь одного элемента ``! А что если нужно получить доступ к дескриптору `<p>`? Можно ли повторно использовать этот код? Определенно нет, поскольку элемент `<p>` находится в другом узле дерева. Потребуется написать похожий код для захвата элемента `<p>` или настроить существующий код с учетом условий поиска этого элемента. Но если внутри дочернего `<div>` были другие дескрипторы ``? Как получить специфичный элемент ``? Решение этих проблем приведет к постоянному росту размера кода, по мере того, как к нему будет добавляться новая условная логика.

Что если затем нужно повторить это упражнение в разных местах, которые имеют слегка отличающуюся структуру? Или же если в будущем разметка немного изменится, модифицируя структуру? Вам придется корректировать все функции с учетом новой иерархии. Очевидно, что если продолжать в таком духе, код довольно скоро станет громоздким и предрасположенным к ошибкам.

Селекторы jQuery помогают устраниТЬ беспорядок. За счет использования предварительно определенных соглашений мы можем совершать обход модели DOM с помощью всего нескольких строк кода. Давайте посмотрим, как эти соглашения сокращают объем кода, необходимого для выполнения тех же действий, что и в предыдущих примерах.

Ниже показано, как можно переписать логику обхода с применением селекторов jQuery. Выбор элемента по идентификатору теперь будет выглядеть так:

```
$("#myDiv").text("Hello jQuery!");
```

Мы вызываем функцию `$()` библиотеки jQuery, передавая ей заранее определенный шаблон. Символ `#` в этом шаблоне обозначает селектор идентификатора, так что шаблон `#myDiv` эквивалентен `document.getElementById("myDiv")`.

После получения ссылки на элемент можно изменить его внутренний текст с помощью метода `text()` библиотеки jQuery. Это подобно установке свойства `innerText`, но не так многословно.

Интересный момент, который здесь необходимо отметить, связан с тем, что почти все методы jQuery возвращают объект jQuery, который представляет собой оболочку вокруг собственного элемента DOM. Эта оболочка позволяет строить цепочки вызовов; например, изменить текст и цвет элемента можно следующим образом:

```
$(".normal > span")           // возвращает объект jQuery
  .contains("Welcome!")        // снова возвращает объект jQuery
  .text("...")                 // и еще раз возвращает объект jQuery
  .css({color: "red"});
```

Из-за того, что каждый вызов `(.text(), .css())` возвращает тот же самый объект jQuery, вызовы могут осуществляться последовательно. Такой стиль построения цепочек вызовов делает код “текучим”, упрощая его чтение, и поскольку код доступа к элементам повторять не требуется, общий объем кода, который придется написать, сокращается.

Подобно шаблону с идентификатором, выбор по имени класса будет иметь такую форму:

```
$(".normal").text("Hello jQuery!");
```

Шаблон для селектора класса выглядит как `".имяКласса"`.



Вспомните, что метод `getElementsByName()` возвращает массив, поэтому в приведенном выше случае jQuery изменит текст во всех элементах в массиве! Таким образом, при наличии множества элементов с одним и тем же именем класса, для получения нужного элемента необходимо применять дополнительные фильтры.

Взгляните, насколько просто обращаться к элементам с указанием отношения “родительский–дочерний” и захватить элемент `` из исходного примера:

```
$(".normal > span").text("Welcome to jQuery!");
```

Символ > указывает отношение *родительский* > *дочерний*. Можно даже произвести фильтрацию на основе контента `` (или любого другого элемента):

```
$(".normal > span").contains("Welcome!").text("Welcome to jQuery!");
```

Вызов `.contains()` отфильтровывает элементы, которые содержат указанный текст. Таким образом, если имеется множество элементов ``, и единственный способ их различения (в случае отсутствия идентификатора, имени класса и т.п.) связан с проверкой контента, то селекторы jQuery это также делают простым.

Библиотека jQuery предлагает намного больше шаблонов селекторов. Чтобы ознакомиться с ними, обратитесь на сайт документации jQuery по адресу:

<http://api.jquery.com/category/selectors>

Реагирование на события

Каждый элемент DOM на HTML-странице способен инициировать события, такие как “щелчок”, “перемещение курсора мыши”, “изменение” и т.д. События являются мощным механизмом для добавления интерактивности к странице: события можно *прослушивать* и выполнять в ответ одно или более действий, совершенствуя пользовательский интерфейс.

Например, предположим, что имеется форма с множеством полей. За счет прослушивания события `onClick` кнопки `Submit` (Отправить) можно выполнить проверку достоверности пользовательского ввода и отобразить любые сообщения об ошибках, не обновляя страницу.

Давайте добавим кнопку, по щелчку на которой будет открываться окно предупреждения с сообщением “hello events!”. В традиционной разметке HTML/JavaScript код будет выглядеть примерно так:

```
<input id="helloButton" value="Click Me" onclick="doSomething();">
<script type="text/javascript">
    function doSomething() {
        alert("hello events!");
    }
</script>
```

Обработчик события `onclick` (или *прослушиватель*) указан в разметке: `onclick="doSomething();"`. После щелчка на этой кнопке показанный выше код отображает окно предупреждения с сообщением “hello events!”.

Присоединить обработчики событий можно также и по-другому:

```
<input id="helloButton" value="Click Me">
<script type="text/javascript">
    function doSomething() {
        alert("hello events!");
    }
    document.getElementById("helloButton").onclick = doSomething;
</script>
```

Обратите внимание, что в разметке больше не определяется поведение `onclick`. Здесь мы отделили презентацию от поведения, присоединив поведение за пределами логики презентации. В результате это не только дает более чистый код, но также обеспечивает возможность многократного использования логики презентации и поведения где угодно без необходимости во внесении множества изменений.

Этот очень простой пример демонстрирует функционирование базовой обработки событий. В реальных приложениях JavaScript-функции будут выглядеть более сложными и могут делать намного больше, нежели отображение простого окна с сообщением пользователю.

А теперь взгляните на соответствующий код jQuery для указания обработчиков событий:

```
<input id="helloButton" value="Click Me">  
<script type="text/javascript">  
    function doSomething() {  
        alert("hello events!");  
    }  
    $(function() {  
        $("#helloButton").click(doSomething);  
    });  
</script>
```

Сначала с помощью jQuery получается ссылка на кнопку, используя для этого селектор `$("#helloButton")`, и затем вызывается метод `.click()` для присоединения обработчика события. Конструкция `.click()` на самом деле является сокращением для `.bind("click", handler)`.

`$(function)` – это сокращение, которое сообщает jQuery о необходимости присоединения обработчиков событий после того, как модель DOM загрузится в браузер. Вспомните, что дерево DOM загружается сверху вниз, т.е. браузер загружает каждый элемент по мере того, как встречает его в древовидной структуре.

Браузер запускает событие `window.onload` сразу после завершения разбора дерева DOM и загрузки всех сценариев, таблиц стилей и других ресурсов. При этом `$()` прослушивает указанное событие и выполняет функцию (которая в действительности является обработчиком события), присоединяющую различные обработчики событий элементов.

Другими словами, `$(function() {...})` – это принятый в jQuery способ написания сценариев:

```
window.onload = function() {  
    $("#helloButton").click(function() {  
        alert("hello events!");  
    });  
}
```

Обработчик события можно также указать *встроенным* образом:

```
$(function() {  
    $("#helloButton").click(function() {  
        alert("hello events!");  
    });  
});
```

Интересно отметить, что если не передать какую-то функцию в `.click()`, запускается событие щелчка. Это удобно, когда требуется программно инициировать щелчок на кнопке:

```
$("#helloButton").click(); // отобразит "hello events!"
```

Манипулирование DOM

Библиотека jQuery предлагает простой и мощный механизм для манипулирования DOM, или изменения свойств как самой модели DOM, так и любого ее элемента.

Например, вот как изменить свойства CSS элемента:

```
// Установить красный цвет для текста кнопки.  
$("#helloButton").css("color", "red");
```

Вы уже видели кое-что из этого в действии; вспомните пример с высотой, приведенный ранее в главе:

```
// Возвратить высоту элемента.  
var height = $("#elem").height();
```

В дополнение к простым манипуляциям наподобие показанной, библиотека jQuery позволяет легко создавать, заменять и удалять любую разметку из группы элементов или корня документа.

В следующем примере демонстрируется добавление группы элементов к существующему элементу `<div>`:

```
<div id="myDiv">  
</div>  
  
<script type="text/javascript">  
    $("#myDiv").append("<p>I was inserted <i>dynamically</i></p>");  
</script>
```

Это дает в результате:

```
<div id="myDiv">  
    <p>I was inserted <i>dynamically</i></p>  
</div>
```

Удалить любой элемент (или набор элементов) также очень просто:

```
<div id="myDiv">  
    <p>I was inserted <i>dynamically</i></p>  
</div>  
  
<script type="text/javascript">  
    $("#myDiv").remove("p"); // удалит <p> и его дочерние элементы  
</script>
```

Ниже показан результат выполнения приведенного кода:

```
<div id="myDiv">  
</div>
```

Библиотека jQuery предоставляет несколько методов, предназначенных для управления расположением разметки, которые кратко описаны в табл. 4.1.

Таблица 4.1. Обычно используемые методы манипулирования DOM

Метод	Описание
.prepend()	Выполняет вставку в начале соответствующего элемента
.before()	Выполняет вставку перед соответствующим элементом
.after()	Выполняет вставку после соответствующего элемента
.html()	Заменяет всю HTML-разметку внутри соответствующего элемента

AJAX

AJAX (Asynchronous JavaScript and XML – асинхронный JavaScript и XML) – это технология, которая позволяет странице запрашивать или посыпать данные без необходимости в обновлении или обратной отправке.

Использование асинхронных запросов для доступа к данным в фоновом режиме (или по требованию) существенно улучшает пользовательский интерфейс, т.к. пользователь не должен ждать, пока загрузится полная страница. И поскольку полная страница не должна перегружаться, объем данных, запрашиваемых у сервера, значительно уменьшается, что дает в результате более короткое время отклика.

Центральной частью AJAX является объект XMLHttpRequest, который первоначально был разработан Microsoft для использования в Outlook Web Access под управлением Exchange Server 2000. Довольно скоро он был принят ведущими игроками отрасли, такими как Mozilla, Google и Apple, а сейчас поддерживается стандартом W3C (<http://www.w3.org/TR/XMLHttpRequest/>).

Типичный запрос AJAX с применением объекта XMLHttpRequest выглядит следующим образом:

```
// Создать объект XMLHttpRequest.  
var xhr = new XMLHttpRequest();  
  
// Открыть новый запрос GET для извлечения домашней страницы google.com.  
xhr.open("GET", "http://www.google.com/", false);  
  
// Отправить запрос без какого-либо контента (null).  
xhr.send(null);  
  
if (xhr.status === 200) { // Код состояния HTTP 200 указывает на успешный запрос.  
    // Вывести текст ответа на консоль браузера (Firefox, Chrome, IE 8+).  
    console.log(xhr.responseText);  
}  
else { // Что-то пошло не так, сохранить в журнале сведения об ошибке.  
    console.log("Error occurred: ", xhr.statusText);  
}
```

В этом примере создается *синхронный* запрос (третий параметр в `xhr.open()`), а это означает, что браузер *приостановит* выполнение сценария до тех пор, пока не будет возвращен ответ. Как правило, таких асинхронных запросов AJAX стараются избегать любой ценой, поскольку веб-страница не будет реагировать вплоть до получения ответа, давая в результате неудобный пользовательский интерфейс.

К счастью, переключиться от синхронного запроса на асинхронный очень легко: нужно просто установить третий параметр `xhr.open()` в `true`. Теперь по причине асинхронной природы браузер не будет останавливаться; он выполнит следующую строку (с проверкой `xhr.status`) незамедлительно. Скорее всего, это приведет к сбою, потому что запрос мог еще не завершить свое выполнение.

Чтобы справиться с такой ситуацией, потребуется указать *обратный вызов* – функцию, которая вызывается, когда запрос обработан и ответ получен.

Ниже приведен модифицированный код:

```
// Создать объект XMLHttpRequest.  
var xhr = new XMLHttpRequest();  
  
// Открыть новый запрос GET для извлечения домашней страницы google.com.  
xhr.open("GET", "http://www.google.com/", true);
```

```

// Присоединить обратный вызов, который будет инициирован,
// как только обработается запрос.
xhr.onreadystatechange = function (evt) {
    // По мере продвижения запроса по различным этапам обработки
    // значение readyState будет изменяться.
    // Эта функция вызывается каждый раз, когда readyState изменяется,
    // так что readyState === 4 проверяет, завершена ли обработка.
    if (xhr.readyState === 4) {
        if (xhr.status === 200) {
            console.log(xhr.responseText)
        }
        else {
            console.log("Error occurred: ", xhr.statusText);
        }
    }
};

// Отправить запрос без какого-либо контента (null).
xhr.send(null);

```

Этот код в основном идентичен синхронной версии, за исключением того, что в нем определена функция обратного вызова, которая выполняется всегда, когда сервер посыпает обратно любую информацию.



Все обратные вызовы должны присоединяться *перед* вызовом `xhr.send()`, иначе инициироваться они не будут.

Давайте рассмотрим эквивалентный код jQuery. Библиотека jQuery предлагает метод `.ajax()` и разнообразные сокращения для решения распространенных задач с использованием AJAX.

Вот версия jQuery:

```

$.ajax("google.com")      // Выдать запрос GET для извлечения домашней
                          // страницы google.com.
    .done(function(data) { // Обработчик успешного исхода (код состояния 200).
        console.log(data);
    })
    .fail(function(xhr) { // Обработчик ошибок (код состояния, отличный от 200).
        console.log("Error occurred: ", xhr.statusText);
    });

```

В первой строке кода указывается URL, с которого необходимо запросить данные. Затем задаются функции обратного вызова для удачного исхода и ошибочных условий (jQuery самостоятельно позаботится о проверке `readyState` и кода состояния).

Обратите внимание, что мы не обязаны указывать тип запроса (GET) и то, является ли он асинхронным. Причина в том, что по умолчанию jQuery использует GET, а `$.ajax()` является асинхронным.

Эти (и другие) параметры можно переопределить для более точной настройки запроса:

```

$.ajax({
    url: "google.com",
    async: true,           // Значение false сделает запрос синхронным.
    type: "GET",          // GET или POST (по умолчанию принимается GET).
    done: function(data) { // Обработчик успешного исхода (код состояния 200).
        console.log(data);
    },
}

```

```
        fail: function(xhr) { // Обработчик ошибок (код состояния, отличный от 200).
            console.log("Error occurred: ", xhr.statusText);
        }
    });
});
```

Запрос jQuery AJAX (<http://api.jquery.com/jQuery.ajax>) поддерживает множество других параметров, кроме показанных здесь. За подробными сведениями обращайтесь на сайт документации jQuery.



Методы `.done()` и `.fail()` появились в версии jQuery 1.8. Если вы работаете с более старой версией jQuery, применяйте вместо них `.success()` и `.error()`.

Проверка достоверности на стороне клиента

В главе 3 были представлены приемы проверки достоверности на стороне сервера. В этом разделе вы увидите, как можно улучшить удобство работы пользователей, выполняя часть такой проверки достоверности прямо на стороне клиента (не обращаясь к серверу) с помощью библиотеки jQuery и подключаемого модуля проверки достоверности jQuery.

Инфраструктура ASP.NET MVC (начиная с версии 3) предлагает ненавязчивую проверку достоверности на стороне клиента, полностью готовую к применению. Клиентская проверка достоверности по умолчанию включена, но ее легко отключить или снова включить, настраивая следующие два параметра в файле `web.config`:

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

Важной особенностью выполнения клиентской проверки с помощью подключаемого модуля проверки достоверности jQuery является возможность использования атрибутов `DataAnnotation`, определенных в модели, а это означает, что для начала ее применения понадобится сделать не так уж много.

Давайте вернемся к модели `Auction` из главы 3, чтобы посмотреть, как аннотации данных использовались при проверке достоверности ввода:

```
public class Auction
{
    [Required]
    [StringLength(50,
        ErrorMessage = "Title cannot be longer than 50 characters")]
    public string Title { get; set; }

    [Required]
    public string Description { get; set; }

    [Range(1, 10000,
        ErrorMessage = "The auction's starting price must be at least 1")]
    public decimal StartPrice { get; set; }

    public decimal CurrentPrice { get; set; }

    public DateTime EndTime { get; set; }
}
```

А вот представление, которое визуализирует сообщения проверки достоверности:

```
<h2>Create Auction</h2>
@using (Html.BeginForm())
{
    @Html.ValidationSummary()
    <p>
        @Html.LabelFor(model => model.Title)
        @Html.EditorFor(model => model.Title)
        @Html.ValidationMessageFor(model => model.Title, "*")
    </p>
    <p>
        @Html.LabelFor(model => model.Description)
        @Html.EditorFor(model => model.Description)
        @Html.ValidationMessageFor(model => model.Description, "*")
    </p>
    <p>
        @Html.LabelFor(model => model.StartPrice)
        @Html.EditorFor(model => model.StartPrice)
        @Html.ValidationMessageFor(model => model.StartPrice)
    </p>
    <p>
        @Html.LabelFor(model => model.EndTime)
        @Html.EditorFor(model => model.EndTime)
        @Html.ValidationMessageFor(model => model.EndTime)
    </p>
    <p>
        <input type="submit" value="Create" />
    </p>
}
```

Выполняемая здесь проверка довольно проста, но при проверке достоверности на стороне сервера страница должна быть передана серверу через обратную отправку, пользовательский ввод должен быть проверен на сервере и в случае возникновения ошибок клиенту должны быть отосланы сообщения, которые после полного обновления страницы отобразятся пользователю.

При проверке достоверности на стороне клиента пользовательский ввод проверяется перед отправкой формы, поэтому никакой обратной отправки серверу и обновления страницы не происходит, а результаты проверки немедленно показываются пользователю!

Чтобы реализовать проверку достоверности на стороне клиента, добавьте в представление ссылку на сценарии подключаемого модуля проверки достоверности jQuery:

```
<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>
```



Если для генерации представлений Create (Создание) или Edit (Редактирование) используется мастер добавления представлений (Add View) среди Visual Studio, можно отметить флажок Reference script libraries (Ссылаться на библиотеки сценариев), заставив Visual Studio добавлять эти ссылки автоматически. Однако вместо реализации ссылок посредством дескрипторов `<script>`, как было показано выше, Visual Studio сделает это через ссылку на пакет сценариев `~/bundles/jquery-val` в конце представления. Дополнительные сведения о пакетировании сценариев смите в разделе “Пакетирование и минимизация” главы 13.

Если теперь запустить приложение и просмотреть исходный код страницы создания аукционного товара (через пункт View Source (Просмотреть исходный код) в контекстном меню), можно увидеть следующую разметку (в которой включены ненавязчивый JavaScript и проверка достоверности на стороне клиента):

```
<form action="/Auctions/Create" method="post" novalidate="novalidate">
    <div class="validation-summary-errors" data-valmsg-summary="true">
        <ul>
            <li>The Description field is required.</li>
            <li>The Title field is required.</li>
            <li>Auction may not start in the past</li>
        </ul>
    </div>
    <p>
        <label for="Title">Title</label>

        <input class="input-validation-error"
               data-val="true"
               data-val-length="Title cannot be longer than 50 characters"
               data-val-length-max="50"
               data-val-required="The Title field is required."
               id="Title" name="Title" type="text" value="">

        <span class="field-validation-error"
              data-valmsg-for="Title"
              data-valmsg-replace="false">*</span>
    </p>
    <p>
        <label for="Description">Description</label>

        <input class="input-validation-error"
               data-val="true"
               data-val-required="The Description field is required."
               id="Description" name="Description" type="text" value="">

        <span class="field-validation-error"
              data-valmsg-for="Description"
              data-valmsg-replace="false">*</span>
    </p>
    <p>
        <label for="StartPrice">StartPrice</label>

        <input data-val="true"
               data-val-number="The field StartPrice must be a number."
               data-val-range="The auction's starting price must be at least 1"
               data-val-range-max="10000"
               data-val-range-min="1"
               data-val-required="The StartPrice field is required."
               id="StartPrice" name="StartPrice" type="text" value="0">

        <span class="field-validation-valid"
              data-valmsg-for="StartPrice"
              data-valmsg-replace="true"></span>
    </p>
    <p>
        <label for="EndTime">EndTime</label>

        <input data-val="true"
               data-val-date="The field EndTime must be a date."
               id="EndTime" name="EndTime" type="text" value="">
    
```

```
<span class="field-validation-valid"
      data-valmsg-for="EndTime"
      data-valmsg-replace="true"></span>
</p>
<p>
    <input type="submit" value="Create">
</p>
</form>
```

Когда включены ненавязчивый JavaScript и проверка достоверности на стороне клиента, ASP.NET MVC визуализирует критерий проверки и соответствующие сообщения в виде атрибутов `data-val`. Подключаемый модуль проверки достоверности jQuery будет применять эти атрибуты для выяснения правил проверки и необходимых сообщений об ошибках, предназначенных для отображения пользователю в случае нарушения этих правил.

Попробуйте отправить форму с какими-нибудь недопустимыми значениями. Вы увидите, что форма в действительности не была отправлена, а вместо этого отобразились сообщения об ошибках рядом с полями, содержащими недопустимые значения.

“За кулисами” подключаемый модуль проверки достоверности jQuery присоединяется к форме обработчик события `onsubmit`. После отправки формы этот подключаемый модуль проходит по всем полям ввода и проверяет их на предмет ошибок по заданному критерию. В случае нахождения ошибки отображается соответствующее сообщение.

Будучи по своей природе ненавязчивым, подключаемый модуль проверки достоверности jQuery не выдает никакого кода для страницы и не требует с вашей стороны какого-либо связывания логики проверки достоверности клиентской стороны с событиями страницы. Вместо этого код присоединяется к событию `onsubmit`, а также логика проверки достоверности является частью файлов `jquery.validate.js` и `jquery.validate.unobtrusive.js`.

Хороший аспект ненавязчивости заключается в том, что если вы забудете включить эти два сценария, страница по-прежнему будет визуализироваться без ошибок, но только проверка достоверности не будет происходить на стороне клиента.

Этот раздел был предназначен для того, чтобы продемонстрировать, насколько легко приступить к использованию проверки достоверности клиентской стороны, сохраняя ее простой и минимально специализированной. Подключаемый модуль проверки достоверности jQuery является довольно сложным и предлагает множество других возможностей и настроек. Обязательно почитайте о нем на официальной странице документации по адресу <http://docs.jquery.com/Plugins/Validation>.

Резюме

Библиотека jQuery существенно упрощает межбраузерную разработку. Готовая поддержка jQuery в ASP.NET MVC означает возможность быстрого построения насыщенных и высоко-интерактивных пользовательских интерфейсов клиентской стороны с написанием весьма небольшого количества строк кода. Благодаря проверке достоверности на стороне клиента и ненавязчивому JavaScript, проверка пользовательского ввода может осуществляться с минимальными усилиями. Комбинирование всех этих приемов помогает упростить разработку высокого-интерактивных веб-приложений.

Переход на следующий уровень

В этой части...

Глава 5. Архитектура веб-приложений

Глава 6. Улучшение сайта с помощью AJAX

Глава 7. ASP.NET Web API

Глава 8. Расширенная работа с данными

Глава 9. Безопасность

Глава 10. Разработка веб-приложений для мобильных
устройств

Архитектура веб-приложений

В первых нескольких главах этой книги был представлен ряд ключевых концепций, которые важны для понимания того, как использовать ASP.NET MVC Framework. Эта глава построена на упомянутых ключевых концепциях. В ней подробно рассматриваются фундаментальные шаблоны проектирования и принципы, заложенные в инфраструктуру ASP.MVC Framework, а также их применение при создании веб-приложений ASP.NET MVC.

Шаблон “модель-представление-контроллер”

Шаблон “модель-представление-контроллер” (Model-View-Controller – MVC) – это архитектурный шаблон пользовательского интерфейса, который обеспечивает разделение ответственности среди множества уровней приложения. Вместо размещения всей бизнес-логики и кода доступа к данным для приложения в одном месте, MVC стимулирует разделение логики приложения на специфические классы, каждый из которых имеет небольшой специальный набор ответственостей.

Шаблон MVC не является новой концепцией и определенно не относится к специфике платформы .NET Framework или даже веб-разработки; на самом деле он был создан программистами в Xerox PARC в конце 1970-х годов и применялся к приложениям, для написания которых использовался язык программирования SmallTalk. С тех пор многие считают MVC одним из наименее достойно оцененных шаблонов, созданных до настоящего времени. Причина в том, что MVC представляет собой очень высокую-ровневый шаблон, который породил многие связанные реализации и подшаблоны.

Разделение ответственности

Разделение ответственности – это принцип вычислительной техники, который стимулирует разнесение полномочий для приложения или сценария использования по множеству компонентов, при этом каждый компонент получает отличающийся небольшой набор ответственостей. Под “ответственностью” можно понимать специфичное множество функциональных возможностей или поведений. Разделение ответственности традиционно достигается применением инкапсуляции и абстрагирования для лучшей изоляции одного набора ответственостей от другого. Например, разделение ответственности может быть применено к архитектуре приложения за счет разнесения различных уровней приложения по уровням презентации, бизнес-логики и логики доступа к данным, каждый из которых является логически и физически отдельным.

Разделение ответственности – мощный принцип, часто используемый при проектировании инфраструктур и платформ. К примеру, веб-страницы в период становления веб-сети содержали разметку, скомбинированную с компоновкой, стилем и даже данными, в одном и том же документе. За эти годы появились стандарты, основанные на разделении ответственности, и то, что когда-то было одним документом, теперь разбивается на три части: HTML-документ, который в основном сосредоточен на структуре контента; одна или более таблиц стилей CSS, определяющих общее оформление документа; и код JavaScript для подключения набора поведений к документу.

В контексте шаблона MVC разделение ответственности применяется для определения полномочий ключевых компонентов: модели, представления и контроллера. На рис. 5.1 показано взаимодействие между различными компонентами MVC и основная их ответственность.

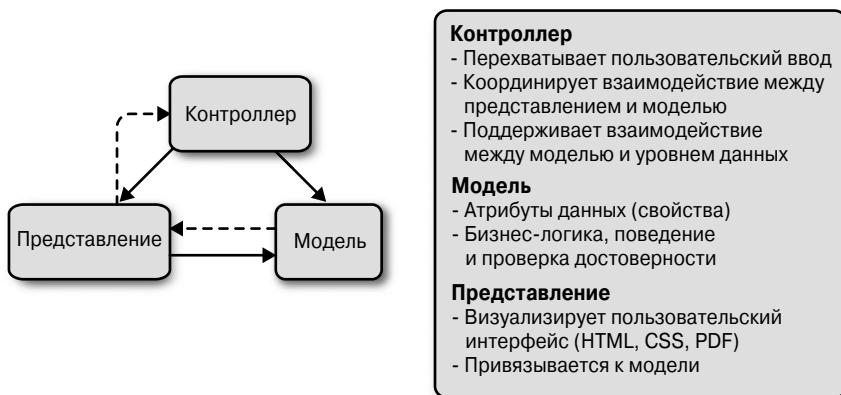


Рис. 5.1. Разделение ответственности в MVC

Важно отметить, что хотя каждый индивидуальный компонент имеет собственный набор ответственостей, компоненты могут и должны полагаться друг на друга. Например, в рамках MVC контроллеры отвечают за извлечение данных из модели и синхронизацию изменений между представлением и моделью. Контроллер может быть ассоциирован с одним и более представлениями. Каждое из таких представлений отвечает за отображение данных определенным способом, но при обработке и извлечении этих данных полагается на контроллер. Хотя одни компоненты могут довольно сильно зависеть от других компонентов, для каждого компонента критически важно сконцентрироваться на собственных ответственостях и оставить прочие ответственности другим компонентам.

MVC и веб-платформы

Первоначальный шаблон MVC был разработан с учетом предположения, что представление, контроллер и модель находятся внутри одного и того же контекста. Шаблон в значительной мере полагается на то, что каждый компонент способен напрямую взаимодействовать с другими и разделять состояние между пользовательскими взаимодействиями. Например, контроллеры будут использовать шаблон проектирования *Observer* (Наблюдатель) для отслеживания изменений в представлении и реагирования на пользовательский ввод. Такой подход хорошо работает, когда контроллер, представление и модель существуют в рамках одного и того же контекста памяти.

В веб-приложении концепция состояния не поддерживается, и представление (HTML) запускается на стороне клиента внутри браузера. Контроллер не может пользоваться шаблоном проектирования Observer для мониторинга изменений; вместо этого из представления в контроллер должен быть отправлен HTTP-запрос. Для решения этой проблемы был задействован шаблон проектирования *Front Controller* (Контроллер входа/Единая точка входа), показанный на рис. 5.2. Главная концепция, лежащая в основе этого шаблона, заключается в том, что когда HTTP-запрос отправлен, контроллер его перехватывает и обрабатывает. Контроллер отвечает за определение того, как обрабатывать этот запрос, и за отправку результатов обратно клиенту.

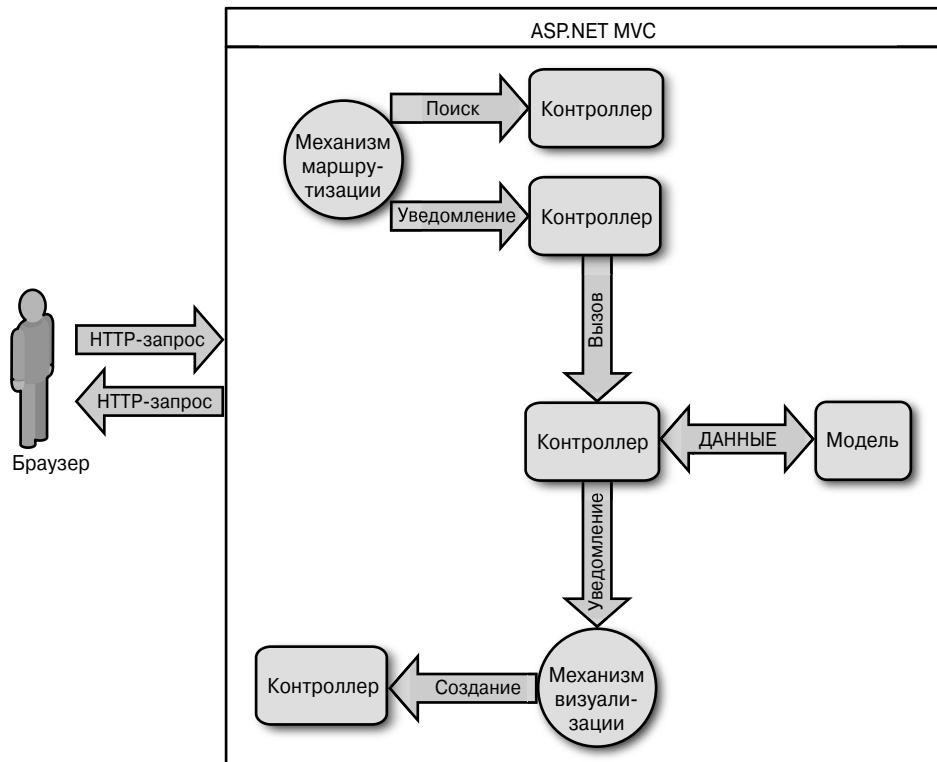


Рис. 5.2. Шаблон проектирования Front Controller

Мощь шаблона проектирования *Front Controller* становится очевидной, если учесть, что современные веб-приложения часто выполняют одну и ту же логику для множества запросов, но потенциально возвращают отличающийся контент для каждого отдельного запроса. Например, одно и то же действие контроллера может обрабатывать и обычный запрос браузера, и запрос AJAX, однако при этом браузер ожидает полностью визуализированную веб-страницу (с компоновкой, таблицами стилей, сценариями и т.д.), а запрос AJAX может получить частичное HTML-представление или даже низкоуровневые данные JSON. Во всех указанных случаях логика контроллера остается той же самой, и представление может оставаться в неведении, откуда поступили данные.

В приложении ASP.NET MVC механизмы маршрутизации и представлений участвуют в обработке HTTP-запроса. Когда получен URI запроса (к примеру,

/auctions/detail/25), исполняющая среда ASP.NET MVC ищет его в таблице маршрутов и вызывает соответствующее действие контроллера. Контроллер обрабатывает запрос и определяет тип возвращаемого результата. Возвратив результат представления, ASP.NET MVC Framework делегирует загрузку и визуализацию соответствующего запрошенному представлению механизму представлений.

Разработка архитектуры веб-приложения

Положенное в основу ASP.NET MVC Framework проектное решение управляется принципом разделения ответственности. В дополнение к механизмам маршрутизации и представлений, инфраструктура предлагает пользоваться фильтрами действий, которые служат для поддержки сквозной функциональности, такой как безопасность, кеширование и обработка ошибок. При проектировании и разработке архитектуры веб-приложения ASP.NET MVC важно понимать, как инфраструктура использует этот принцип и каким образом спроектировать свое приложение так, чтобы эффективно задействовать его.

Логическое проектирование

Логическая (концептуальная) архитектура приложения сосредоточена на отношениях и взаимодействиях между компонентами, и такие компоненты сгруппированы в логические уровни, которые поддерживают специфические наборы функциональных возможностей.

Компоненты должны быть спроектированы для обеспечения принципа разделения ответственности и применять абстракцию для взаимодействия между компонентами. Сквозная функциональность, подобная безопасности, ведению журнала и кешированию, должна быть изолирована в виде различных служб приложения. Эти службы должны поддерживать подход подключаемых модулей. Переключение между разными типами аутентификации или реализация других источников журналов не должна оказывать влияние на другие части приложения.

Логическое проектирование веб-приложения ASP.NET MVC

Инфраструктура ASP.NET MVC Framework была разработана для поддержки такого типа логического проектирования. В дополнение к изолированию представления, контроллера и модели, инфраструктура включает несколько фильтров действий, которые обрабатывают различные типы сквозной функциональности, и множество типов результатов действий для представлений, JSON, XML и частичных страниц. Поскольку инфраструктура обладает буквально бесконечной расширяемостью, разработчики могут создавать и подключать собственные специальные фильтры действий и результаты.

Примером специального типа `ActionFilter` является `SingleSignOnAttribute`, который был создан для поддержки аутентификации единого входа (Single Sign On Authentication) в рамках множества веб-приложений ASP.NET:

```
public class SingleSignOnAttribute : ActionFilterAttribute, IActionFilter
{
    void OnActionExecuted(ActionExecutedContext filterContext)
    {
        // Проверить маркер безопасности и аутентифицировать пользователя.
    }
}
```

```

void OnActionExecuting(ActionExecutingContext filterContext)
{
    // Код предварительной обработки, применяемый для проверки,
    // когда маркер безопасности существует.
}
}

```

Лучший способ взаимодействия с логическим проектом приложения предусматривает создание визуальной презентации каждого компонента и соответствующего ему уровня. На рис. 5.3 показан типовой логический проект веб-приложения ASP.NET MVC. Обратите внимание на разнесение сквозной функциональности по разным службам приложения.

Элементы, изображенные на рис. 5.3, описаны в табл. 5.1.

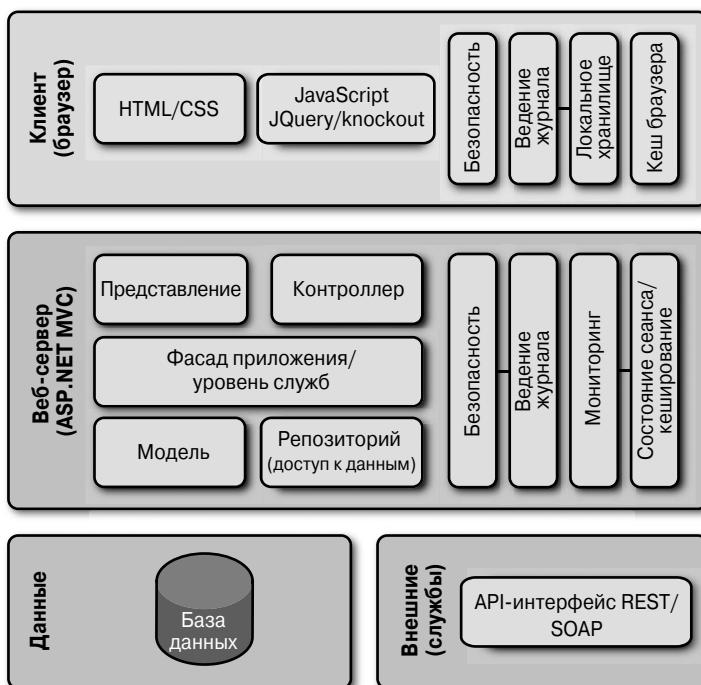


Рис. 5.3. Логическая архитектура веб-приложения

Таблица 5.1. Описание компонентов

Название	Уровень	Описание
HTML/CSS	Клиент	Элементы пользовательского интерфейса, применяемые для описания компоновки и стиля приложения
JavaScript	Клиент	Любая логика клиентской стороны, используемая для проверки достоверности и бизнес-обработки
Безопасность	Клиент	Маркер безопасности (cookie-набор)
Ведение журнала	Клиент	Локальная служба, применяемая для ведения журнала и мониторинга

Название	Уровень	Описание
Локальное хранилище	Клиент	Локальное хранилище HTML 5 (используемое для кеширования/оффлайнового хранения)
Кеш браузера	Клиент	Кеш, предоставляемый браузером, который применяется для сохранения HTML-разметки, CSS-стилей, изображений и т.д.
Представление	Веб-сервер	Представление серверной стороны, используемое для визуализации HTML-разметки
Контроллер	Веб-сервер	Контроллер приложения, обрабатывающий пользовательский ввод и оснастку
Модель	Веб-сервер	Коллекция классов, представляющих модель предметной области для приложения
Уровень служб	Веб-сервер	Уровень служб, предназначенный для инкапсуляции сложных бизнес-процессов и постоянства
Репозиторий	Веб-сервер	Компоненты доступа к данным (объектно-реляционный отображатель)
Безопасность	Веб-сервер	Служба безопасности, применяемая для аутентификации и авторизации пользователей
Ведение журнала	Веб-сервер	Служба приложения, используемая для ведения журнала
Мониторинг	Веб-сервер	Служба приложения, применяемая для мониторинга работоспособности
Сеанс/кеширование	Данные	Служба приложения, используемая для управления транзитным состоянием
Внешняя служба	Данные	Любые внешние системы, от которых зависит приложение

Полезные советы по логическому проектированию

Многоуровневый проект, такой как показанный на рис. 5.3, обеспечивает наиболее гибкую архитектуру приложения. Каждый уровень имеет дело со специфическим набором ответственостей, и уровни зависят только от уровней, находящихся ниже в стеке. Например, репозиторий для доступа к данным расположен на том же уровне, что и модель, поэтому для него вполне приемлемо иметь зависимость. Модель изолирована от лежащего в основе хранилища данных; ее не заботит, каким образом репозиторий поддерживает постоянство, и даже то, хранится он в локальном файле или в базе данных.

При разработке архитектуры веб-приложения обычно возникает дискуссия по поводу того, где применять бизнес-правила и правила проверки достоверности. Шаблон MVC устанавливает, что модель должна быть ответственной за бизнес-логику. Это верно, хотя в распределенном приложении каждый уровень должен разделять определенную степень ответственности за проверку достоверности пользовательского ввода. В идеальном случае ввод всегда должен быть проверен перед передачей на другой уровень.

Каждый уровень должен принять ответственность за степень проверки достоверности, которую он реализует. Нисходящие уровни никогда не должны предполагать, что вызывающий уровень выполнил все проверки. На стороне клиента должны применяться средства JavaScript (jQuery) для проверки обязательных полей и ограничений ввода в общих элементах управления пользовательского интерфейса (числовых, даты и времени и т.д.). Бизнес-модель приложения должна обеспечивать применение всех бизнес-правил и правил проверки достоверности, в то время как уровень базы данных должен использовать строго типизированные поля и применять ограничения, предотвращая нарушение отношений внешнего ключа.

Что необходимо, так это избегать дублирования бизнес-логики в рамках каждого уровня. Если на экране для администратора предусмотрена специальная логика или дополнительные возможности по сравнению с обычным пользователем, бизнес-модель должна идентифицировать, какие средства включать или отключать, и предоставить флаг для сокрытия или отключения административных полей в случае работы обычного пользователя.

Физическое проектирование

Роль физического проектирования архитектуры заключается в определении физических компонентов и модели развертывания для веб-приложения. Большинство веб-приложений основано на *многоуровневой* модели. Клиентский уровень состоит из HTML-разметки, CSS-стилей и JavaScript-кода, которые выполняются внутри веб-браузера. Клиент выдает HTTP-запросы для извлечения HTML-контента напрямую или выполняет запрос AJAX (который возвращает частичную HTML-страницу, XML-разметку или данные JSON). Уровень приложения включает инфраструктуру ASP.NET MVC Framework (выполняющуюся под управлением веб-сервера IIS) и любые специальные сборки или сборки от независимых поставщиков, используемые приложением. Уровень данных может содержать одну и более реляционных баз данных или баз данных NoSQL, либо одну и более внешних веб-служб SOAP или REST, либо другие API-интерфейсы от независимых разработчиков.

Пространства имен проекта и имена сборок

Перед тем как веб-приложение ASP.NET MVC можно будет развернуть, разработчику необходимо решить, каким образом физически разделить код приложения на разные пространства имен и сборки. При проектировании приложения ASP.NET MVC можно применять множество различных подходов. Разработчик может принять решение хранить все компоненты приложения внутри сборки веб-сайта или же разнести компоненты по разным сборкам. В большинстве случаев имеет смысл помещать уровень бизнес-логики и уровень доступа к данным в разные сборки, отличные от сборки веб-сайта. Обычно это делается для лучшей изоляции бизнес-модели от пользовательского интерфейса и для упрощения написания автоматизированного теста, сосредоточенного на основной логике приложения. Вдобавок, применение этого подхода делает возможным повторное использование уровня бизнес-логики и уровня доступа к данным в других приложениях (консольных приложениях, веб-сайтах, веб-службах и т.д.).

Общее корневое пространство имен (например, компания.*{ИмяПриложения}*) должно единообразно применяться во всех сборках. Каждая сборка должна иметь уникальное подпространство имен, которое соответствует имени сборки. На рис. 5.4 показана структура проекта для образцового приложения Ebuy. Функциональность для этого приложения разделена на три проекта: Ebuy.WebSite включает представление, контроллеры и другие связанные с веб файлы; Ebuy.Core содержит бизнес-

модель для приложения; CustomExtensions состоит из специальных расширений, используемых приложением для привязки модели, маршрутизации и контроллеров. В дополнение предусмотрены два проекта тестирования (не показаны): UnitTests и IntegrationTests.

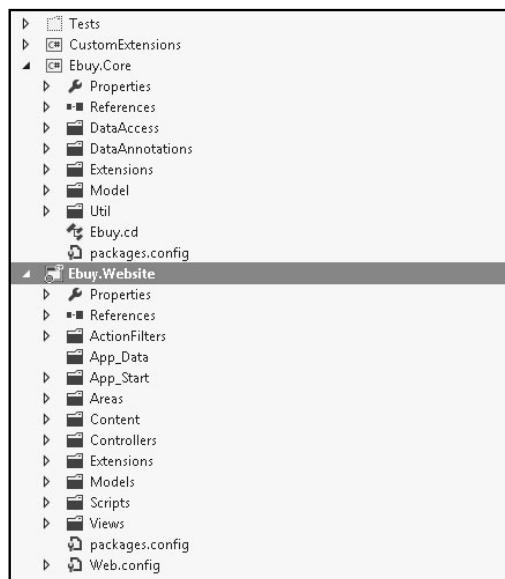


Рис. 5.4. Структура проектов Visual Studio

Варианты развертывания

После создания решения Visual Studio и определения структуры проектов разработчик может воспользоваться автоматизированным или ручным процессом для развертывания скомпилированного приложения на веб-сервере. За дополнительными деталями по развертыванию веб-приложений ASP.NET MVC обращайтесь в главу 19.

На рис. 5.5 показано веб-приложение ASP.NET MVC, которое было сконфигурировано для использования многосерверной веб-фермы и кластеризованной базы данных SQL Server. Веб-приложения ASP.NET MVC поддерживают все разновидности моделей развертывания. Приложение может быть самодостаточным и работать с локальной базой данных SQL Server Express или же применять многоуровневую производственную модель.

Полезные советы по физическому проектированию

Никаких “абсолютно правильных” решений по разработке архитектуры веб-приложения не существует, и с каждым вариантом выбора связаны компромиссы. Важно, чтобы проект приложения был гибким и включал подходящий мониторинг, данные реального времени которого позволили бы принимать обоснованные решения по настройке приложения. Это одна из тех областей, где инфраструктура ASP.NET MVC Framework выгодно выделяется. Она спроектирована с учетом гибкости и расширяемости. Инфраструктура позволяет легко воспользоваться встроенными службами IIS и ASP.NET, и она предлагает высокий уровень расширяемости и возможность подключения различных компонентов и служб.

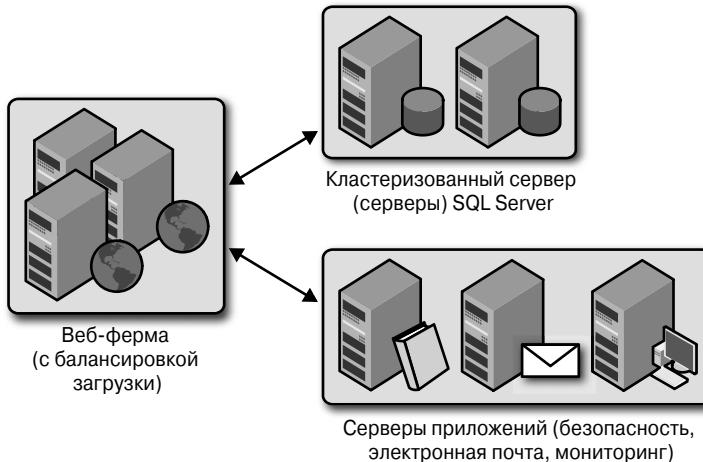


Рис. 5.5. Физический проект веб-приложения

Существует множество факторов, которые необходимо принимать во внимание во время проектирования веб-приложения. Наиболее важными четырьмя факторами являются производительность, масштабируемость, пропускная способность и время ожидания. Выбор, сделанный в пользу одного или нескольких этих факторов, может оказывать влияние на остальные факторы. Настройка стратегии мониторинга является хорошим способом адекватной оценки работоспособности приложения, особенно под нагрузкой, и определения подходящего фактора балансировки.

Производительность и масштабируемость

Выбор в пользу одной лишь производительности либо масштабируемости может весьма значительно повлиять на другой фактор в этой паре. Если проект приложения требует сохранения в кеше большого объема данных, это может повлиять на требования по использованию памяти в приложении. Рабочий процесс IIS должен быть сконфигурирован с учетом требований по использованию памяти: если выделено слишком много памяти, рабочий процесс будет утилизирован. Значительное влияние может также оказывать понимание того, как сборщик мусора .NET освобождает ресурсы. Постоянная загрузка крупной коллекции или набора данных в состояние сеанса может привести к перемещению этого объекта в поколение 2 сборщика мусора .NET или к большему размеру кучи.

Применение веб-фермы является хорошим способом увеличения масштабируемости веб-приложения. Очень важно, чтобы это было учтено при проектировании приложения. Использование фермы влияет на то, как приложение обрабатывает переходное состояние. Если это поддерживается аппаратным и программным обеспечением балансировки загрузки, можно применить подход “липкого” сеанса, который обеспечивает перенаправление пользователя на сервер, инициировавший первоначальный сеанс.

Альтернативный подход заключается в использовании сервера состояния сеансов или сохранения состояния сеанса в базе данных. Всегда имеет смысл сводить к минимуму применение состояния сеанса и обязательно иметь определенные правила тайм-аутов для кэшированных данных.

Поскольку местоположение кэширования состояния сеанса и данных является конфигурируемым, важно удостовериться, что все классы, которые могут использоваться,

корректно настроены для сериализации. Это можно делать за счет применения .NET-атрибута `SerializableAttribute`, реализации интерфейса `ISerializeable` либо использования сериализации контрактов данных Windows Communication Foundation (WCF) или какого-то другого метода сериализации, поддерживаемого в .NET.

Пропускная способность и время ожидания

Работа с пропускной способностью и временем ожидания может оказаться чрезвычайно сложной. Время ожидания обычно является фиксированным ограничением; если сервер расположен в Нью-Йорке, а пользователь просматривает сайт из Японии, то может иметь место значительное (потенциально до пяти секунд) время ожидания для каждого запроса. Существует множество мер, которые могут быть предприняты для решения указанной проблемы, в том числе сжатие файлов JavaScript, использование карт изображений и ограничение количества запросов. Пропускная способность, как правило, более изменчива, но она может значительно пострадать, если приложению требуется передавать большие объемы данных по сети. Лучший вариант решения таких проблем заключается в минимизации числа запросов и сохранении размера полезной нагрузки на небольшом уровне. Хорошие стратегии предполагают включение разбиения на страницы для крупных результирующих наборов, отправку по сети только необходимых полей и применение проверки достоверности и кеширования на стороне клиента, где это только возможно.

Принципы проектирования

При проектировании приложения, инфраструктуры или класса важно думать о расширяемости кода, а не просто о лучшем способе реализации начального набора функциональных возможностей. Эта идея положена в основу проекта инфраструктуры ASP.NET MVC Framework. В инфраструктуре повсеместно используются фундаментальные принципы и наилучшие практические решения объектно-ориентированного проектирования.

SOLID

SOLID является аббревиатурой, которая описывает определенный набор принципов разработки приложений, управляющих надлежащим объектно-ориентированным проектированием и разработкой. В случае применения ко всему приложению эти приемы позволяют создавать модульные компоненты, которые могут быть легко протестированы и гибким образом изменены.

SOLID включает описанные ниже принципы.

Принцип единственной ответственности

Принцип единственной ответственности (Single Responsibility Principle – SRP) означает, что объекты должны иметь единственную ответственность, и все их поведение должно быть сосредоточено на этой одной ответственности. Хорошим примером может служить наличие различных контроллеров для разных экранов. Например, контроллер `HomeController` должен содержать только операции, относящиеся к домашней странице, в то время как контроллер `ProductController` должен поддерживать только операции для страниц товаров. Аналогично, представления должны быть сконцентрированы на визуализации пользовательского интерфейса и избегать любой логики доступа к данным.

Показанный ниже класс `ErrorLoggerManager` является распространенным примером класса, который нарушает принцип SRP. Этот класс имеет два метода, один из которых фиксирует сведения об ошибках в журнале событий, а другой записывает информацию об ошибках в файл. Хотя на первый взгляд группирование этих методов вместе может выглядеть вполне безобидным, данный класс в настоящее время имеет слишком много ответственостей; это станет более очевидным после ввода дополнительных методов регистрации ошибок. Общий признак кода, на который следует обращать внимание – это любой класс с именем `xxxManager`. Такой класс с высокой вероятностью имеет слишком много ответственостей.

```
public class ErrorLoggerManager
{
    public void LogErrorToEventLog(Exception e)
    {
        // Код регистрации.
    }

    public void LogErrorToFile(Exception e)
    {
        // Код регистрации.
    }
}
```

Принцип открытости/закрытости

Принцип открытости/закрытости (Open/Closed Principle – OCP) поддерживает компоненты, которые являются *открытыми* для расширения, но *закрытыми* для модификации. Это дополняет принцип SRP, указывая на то, что вместо добавления все большего и большего количества поведений и ответственостей к классу, для расширения его возможностей необходимо унаследовать от него новый класс. Хорошим примером может быть сквозная служба, подобная регистрации сведений об ошибках: вместо добавления возможности регистрации ошибок в базе данных и в файле к одному и тому же классу, потребуется создать абстракцию, от которой могут наследоваться различные методы регистрации. За счет этого внутренняя работа регистрации в базе данных или в файле изолируется от других реализаций.

Взглянув на класс `ErrorLoggerManager` еще раз, должно быть ясно, что он на самом деле нарушает оба принципа, SRP и OCP. В настоящий момент этот класс имеет два метода, которые очень специфичны в своей реализации: `LogErrorToEventLog` для регистрации сведений об ошибках в журнале событий и `LogErrorToFile` для регистрации ошибок в файле. Когда потребуются дополнительные типы регистрации, возможность сопровождения этого класса очень быстро выйдет из-под контроля.

```
public class ErrorLoggerManager
{
    public void LogErrorToEventLog(Exception e)
    {
        // Код регистрации.
    }

    public void LogErrorToFile(Exception e)
    {
        // Код регистрации.
    }
}
```

Ниже представлена обновленная версия класса `ErrorLogger`, спроектированная согласно принципам SRP и OCP. Здесь был введен интерфейс по имени `ILogSource`, который реализуется каждым типом регистрации. Кроме того, были созданы два дополнительных класса: `EventLogSource` и `FileLogSource`. Оба эти класса имеют дело со специфическими типами регистрации. Теперь, если появятся новые типы регистрации, ни один из этих классов изменять не придется.

```
public class ErrorLogger
{
    public void Log(ILogSource source)
    {
        // Код регистрации.
    }
}

public interface ILogSource
{
    LogError(Exception e);
}

public class EventLogSource : ILogSource
{
    public void LogError(Exception e)
    {
        LogError(Exception e);
    }
}

public class FileLogSource : ILogSource
{
    public void LogError(Exception e)
    {
        LogError(Exception e);
    }
}
```

В этот момент может показаться, что хотя данный подход является более чистым, он также требует написания намного большего объекта кода. Несмотря на то что это может быть и так, в примерах настоящей книги будут продемонстрированы многие преимущества, получаемые от слабо связанных компонентов, которые следуют этому шаблону.

Принцип подстановки Барбары Лисков

Принцип подстановки Барбары Лисков (Liskov Substitution Principle – LSP) гласит, что объекты должны легко заменяться экземплярами их подтипов без влияния на поведение и правила, связанные с этими объектами. Например, хотя может показаться интересной идеей иметь общий базовый класс или интерфейс, такой подход может косвенно привести к появлению кода, нарушающего принцип LSP.

Взгляните на интерфейс `ISecurityProvider` и классы, которые его реализуют. Все выглядит хорошо до тех пор, пока `UserController` не вызовет метод `RemoveUser` класса `ActiveDirectoryProvider`. В этом примере удаление пользователей поддерживает только класс `DatabaseProvider`. Одним из способов решения проблемы могло бы быть добавление специфичной для типа логики, что и сделано в настоящий момент в `UserController`. Однако такой подход обладает крупным недостатком – он нарушает принцип LSP. Генерация исключения в этом случае является плохой идеей, приводящей

к появлению специфичного для типа кода. Решить эту проблему можно за счет использования принципа разделения интерфейса, который рассматривается ниже.

```
public interface ISecurityProvider
{
    User GetUser(string name);
    void RemoveUser(User user);
}

public class DatabaseProvider : ISecurityProvider
{
    public User GetUser(string name)
    {
        // Код для добавления нового пользователя.
    }

    public void RemoveUser(User user)
    {
        // Код для сохранения пользователя.
    }
}

public class ActiveDirectoryProvider : ISecurityProvider
{
    public User GetUser(string name)
    {
        // Код для добавления нового пользователя.
    }

    public void RemoveUser(User user)
    {
        // Active Directory не позволяет удалять пользователей.
        throw new NotImplementedException();
    }
}

public class UserController : Controller
{
    private ISecurityProvider securityProvider;

    public ActionResult RemoveUser(string name)
    {
        User user = securityProvider.GetUser(name);
        if (securityProvider is DatabaseProvider) // нарушает принцип LSP
            securityProvider.Remove(user);
    }
}
```

Принцип разделения интерфейса

Принцип разделения интерфейса (Interface Segregation Principle – ISP) поддерживает использование – и в одновременно ограничивает размер – интерфейсов повсюду в приложении. Другими словами, вместо одного интерфейса, который содержит все поведение для объекта, должно существовать множество мелких и более специфичных интерфейсов. Хорошим примером может служить разделение интерфейсов для сериализации и освобождения в .NET. Класс будет реализовывать интерфейсы `ISerializable` и `IDisposable`, в то время как потребитель, который заинтересован в одной лишь сериализации, будет заботиться только о методах, реализованных интерфейсом `ISerializable`.

В приведено ниже коде демонстрируется пример корректного использования принципа ISP. Были созданы два отдельных интерфейса: один (`ISearchProvider`) содержит только методы для поиска, а другой (`IRepository`) определяет методы для сохранения сущности. Обратите внимание, что класс `SearchController` заботится лишь о поведении для поиска и ссылается только на интерфейс `ISearchProvider`. Этот мощный прием можно применять для обеспечения уровня безопасности. Например, предположим, что необходимо позволить всем производить поиск товаров, но предоставить возможность добавления и удаления товаров только администраторам. За счет использования принципа ISP можно сделать так, чтобы класс `SearchController` разрешал анонимным пользователям только искать товары, но не добавлять и удалять их.

```
public interface ISearchProvider
{
    IList<T> Search<T>(Criteria criteria);
}

public interface IRepository<T>
{
    T GetById(string id);
    void Delete(T);
    void Save(T)
}

public class ProductRepository<T : ISearchProvider, IRepository<Product>>
{
    public IList<Product> Search(Criteria criteria)
    {
        // Код поиска.
    }

    public Product GetById(string id)
    {
        // Код доступа к данным.
    }

    public void Delete(Product product)
    {
        // Код доступа к данным.
    }

    public void Save(Product product)
    {
        // Код доступа к данным.
    }
}

public class SearchController : Controller
{
    private ISearchProvider searchProvider;
    public SearchController(ISearchProvider provider)
    {
        this.searchProvider = provider;
    }

    public ActionResult SearchForProducts(Criteria criteria)
    {
        IList<Products> products = searchProvider.Search<Product>(criteria);
        return View(products);
    }
}
```

Принцип инверсии зависимостей

Принцип инверсии зависимостей (Dependency Inversion Principle – DIP) гласит, что компоненты, зависящие друг от друга, должны взаимодействовать через абстракцию, а не напрямую с помощью конкретной реализации. В качестве хорошего примера можно привести контроллер, который полагается на абстрактный класс или интерфейс для взаимодействия с уровнем доступа к данным, как противоположность созданию экземпляра специфичной разновидности самого объекта доступа к данным.

Этот принцип предоставляет несколько преимуществ: использование абстракции позволяет разрабатывать и изменять разные компоненты независимо друг от друга, появляется возможность вводить новые реализации абстракции и существенно упрощается тестирование, т.к. зависимость можно имитировать.

В следующем разделе мы покажем, как специфичная реализация принципа DIP – инверсия управления (Inversion of Control – IoC) – упрощает это за счет применения отдельного компонента для управления созданием и временем жизни данной абстракции.

Приведенный ниже код – это пример DIP, который будет использоваться. Класс SearchController имеет зависимость от интерфейса ISearchProvider. Вместо непосредственного создания экземпляра ProductRepository контроллер пользуется переданным ему экземпляром ISearchProvider. В следующем разделе вы узнаете, как использовать контейнер IoC для управления зависимостями.

```
public class SearchController : Controller
{
    private ISearchProvider searchProvider;
    public SearchController(ISearchProvider provider)
    {
        this.searchProvider = provider;
    }
}
public class ProductRepository : ISearchProvider
{
}
```

Инверсия управления

Теперь, когда вы ознакомились с принципами проектирования SOLID, самое время погрузиться в магию, которая соединяет все эти концепции вместе: *инверсию управления* (Inversion of Control – IoC). Инверсия управления – это принцип проектирования, который поддерживает слабо связанные уровни, компоненты и классы за счет переворачивания потока управления в приложении.

По сравнению с традиционным процедурным кодом, в котором единственная процедура явно управляет потоком между собой и подпрограммами, IoC использует концепцию разделения выполнения кода и кода, специфичного для задачи. Этот подход позволяет разрабатывать различные компоненты приложения независимо друг от друга. Например, в приложении MVC уровни моделей, представлений и контроллеров могут проектироваться и строиться независимым образом.

Двумя популярными реализациями принципа проектирования IoC являются *внедрение зависимостей* (dependency injection) и *локатор служб* (service locator). Обе они используют одну и ту же базовую концепцию центрального контейнера для управления жизненным циклом зависимости. Основное отличие между этими двумя реализаций касается того, как производится доступ к зависимостям.

Локатор служб полагается на то, что вызывающий компонент обратится и запросит зависимость, а внедрение зависимостей работает за счет введения зависимости внутрь класса путем передачи в конструкторе, установки одного из свойств или выполнения одного из методов.

Понятие зависимостей

Понимание различных типов зависимостей и того, как они управляют отношением между зависимостями, критически важно для минимизации сложности приложения. Зависимости поступают во множество форм: одна сборка .NET может иметь одну или более ссылок на другие сборки .NET; контроллер MVC должен быть унаследован от базового класса контроллера ASP.NET MVC; приложение ASP.NET требует для своего размещения веб-сервера IIS.

На рис. 5.6 показано отношение между контроллером и классом репозитория. В этом сценарии контроллер напрямую создает экземпляр класса репозитория. В настоящее время контроллер тесно связан с классом репозитория. Любое изменение в открытом интерфейсе репозитория потенциально влияет на класс контроллера. Например, если разработчик решит изменить стандартный конструктор для класса репозитория, чтобы принимать один или несколько параметров, это изменение повлияет на класс контроллера.

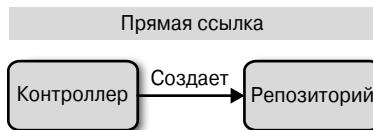


Рис. 5.6. Управление зависимостями

Для того чтобы сделать отношение между контроллером и репозиторием слабо связанным, разработчик может ввести абстракцию в форме интерфейса `IRepository` и переместить создание класса репозитория в фабрику (шаблон). Такая конфигурация показана на рис. 5.7. Теперь контроллер зависит только от интерфейса `IRepository`, и любые изменения в конструкторе класса репозитория не потребуют внесения изменений в класс контроллера. Хотя это устраняет тесное связывание между контроллером и репозиторием, однако вводит новое связывание между контроллером и классом фабрики.

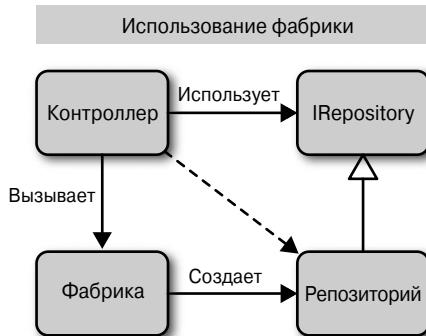


Рис. 5.7. Использование абстракции

На рис. 5.8 видно, как контейнер IoC может быть заменен классом фабрики в качестве средства управления зависимостью между классами контроллера и репозитория. Эта реализация по-прежнему содержит контроллер, использующий в качестве абстракции интерфейс `IRepository`. Но только теперь контроллер ничего не знает о том, каким образом создается репозиторий – контейнер IoC отвечает как за создание, так и за “внедрение” (т.е. передачу) экземпляра репозитория внутрь контроллера. Применение контейнера IoC предоставляет дополнительный уровень функциональности над фабрикой, который заключается в том, что контейнер IoC позволяет конфигурировать управление жизненным циклом класса.

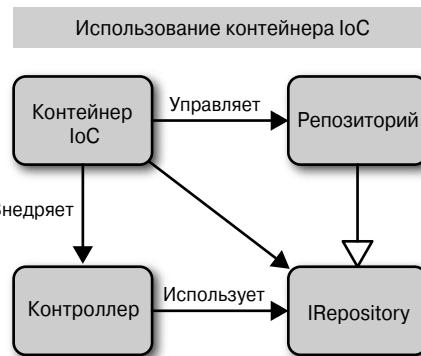


Рис. 5.8. Управление зависимостями с помощью IoC

Локатор служб

Применять шаблон локатора служб довольно легко. Разработчику нужно просто запросить у контейнера IoC специфичный класс службы. Контейнер выясняет, был ли сконфигурирован запрошенный класс, и на основе правил управления жизненным циклом для класса либо создаст его новый экземпляр, либо возвратит ранее созданый экземпляр запрашивающему компоненту.

Шаблон локатора служб работает хорошо, если необходимо напрямую обращаться к отдельной службе в единственном методе или запрашивать службу по имени, а не только по интерфейсу. Основной недостаток использования этого шаблона связан с тем, что код нуждается в прямом доступе к контейнеру IoC, что может ввести тесное связывание между кодом и API-интерфейсом контейнера IoC. Один из способов минимизации такого связывания является абстрагирование от контейнера IoC в пользу специального интерфейса.

Ниже приведен пример класса, который напрямую запрашивает контейнер IoC на предмет специфической службы на основе ее зарегистрированного интерфейса:

```

public class AuctionsController : Controller
{
    private readonly IRepository _repository;
    public AuctionsController()
    {
        IRepository repository = Container.GetService< IRepository >();
        _repository = repository;
    }
}
  
```

Внедрение зависимостей

Шаблон внедрения зависимостей (dependency injection – DI) поддерживает слабо связанный подход, если сравнить его с шаблоном локатора служб. В рамках DI применяется подход, при котором зависимости передаются через конструкторы, свойства или методы. Обычно большинство разработчиков используют внедрение через конструктор, т.к. во многих случаях зависимость требуется сразу же. Тем не менее, определенные контейнеры IoC разрешают подход отложенной (ленивой) загрузки для зависимостей, внедряемых через свойство. В этом случае зависимость не загружается до тех пор, пока не произойдет обращение к свойству.

Класс `AuctionsController` настроен на применение внедрения через конструктор. Этот класс зависит от репозитория, предназначенного для сохранения и извлечения данных, связанных с аукционными товарами; однако контроллер не имеет прямой ссылки на действительную реализацию репозитория. Вместо этого контроллер полагается на интерфейс `IRepository`, а контейнер IoC определяет подходящую реализацию `IRepository` и “внедряет” ее во время выполнения:

```
public class AuctionsController : Controller
{
    private readonly IRepository _repository;
    public AuctionsController(IRepository repository)
    {
        _repository = repository;
    }
}
```

Внедрение зависимостей действительно проявляет себя с лучшей стороны в ситуациях с множеством уровней зависимостей, т.е. когда зависимости сами имеют другие зависимости. Когда контейнер IoC приступает к внедрению зависимости, он проверяет, не загружал ли он ранее экземпляр этой зависимости. Если это не так, контейнер IoC создает новый экземпляр и выясняет, есть ли какие-то зависимости, которые должны быть внедрены. При таком сценарии контейнер IoC обходит дерево зависимостей и создает необходимые зависимости.

В следующем примере показано, как работает соединение зависимостей в цепочку. Когда контейнер IoC создает экземпляр класса `AuctionsController`, он обнаруживает, что этот класс имеет зависимость от `IRepository`. Контейнер проверит, зарегистрирован ли экземпляр `IRepository`, и создаст экземпляр класса `AuctionsRepository`. Поскольку этот класс имеет собственную зависимость, контейнер создаст экземпляр класса `ErrorLogger` и внедрит его в `AuctionsRepository`.

```
public class AuctionsController : Controller
{
    private readonly IRepository _repository;
    public AuctionsController(IRepository repository)
    {
        _repository = repository;
    }
}

public interface IRepository<T>
{
    T GetById(string id);
    void Delete(T);
    void Save(T)
}
```

```

public class AuctionsRepository : IRepository<Auction>
{
    private readonly IErrorLogger _logger;
    public AuctionsRepository(IErrorLogger logger)
    {
        _logger = logger;
    }
    public Auctions GetById(string id)
    {
        // Код доступа к данным.
    }
    public void Delete(Auctions auction)
    {
        // Код доступа к данным.
    }
    public void Save(Auctions auction)
    {
        // Код доступа к данным.
    }
}
public class ErrorLogger : IErrorLogger
{
    public void Log(Exception e)
    {
        // Код регистрации.
    }
}
public interface IErrorLogger
{
    Log(Exception e);
}

```

Выбор контейнера IoC

Используя инверсию управления, разработчик должен иметь в виду пару аспектов: производительность и обработку ошибок. Применение контейнера IoC для управления и внедрения зависимостей может оказаться дорогостоящим. Зависимости должны иметь подходящий жизненный цикл. Если зависимость сконфигурирована как одиночный объект, могут возникнуть проблемы, связанные с многопоточностью, и нужно будет соответствующим образом управлять любыми внешними ресурсами (например, строками подключений). Очень важно избегать использования контейнера IoC для создания крупной коллекции; применять DI для создания коллекции из 1000 элементов – действительно очень плохая идея. Пропущенные или незарегистрированные зависимости могут превратиться в настоящий кошмар при отладке. Разработчик должен отслеживать обязательные зависимости и проверять, что все они зарегистрированы, во время загрузки приложения.

Поскольку большинство контейнеров IoC очень просты, выбор конкретного контейнера обычно зависит более от личных предпочтений разработчика, чем от тщательного сравнения их функциональных возможностей. Для .NET существует ряд доступных контейнеров, каждый из которых предлагает отличающийся подход к внедрению и управлению зависимостями.

Ниже приведен список популярных контейнеров IoC для .NET.

1. Ninject: <http://www.ninject.org>
2. Castle Windsor: <http://www.castleproject.org/container/index.html>
3. Autofac: <http://code.google.com/p/autofac/>
4. StructureMap: <http://structuremap.net/structuremap/index.html>
5. Unity: <http://unity.codeplex.com>
6. MEF: <http://msdn.microsoft.com/ru-ru/library/dd460648.aspx>

Для образцового приложения ЕВиу мы выбрали контейнер Ninject из-за его высокой популярности в сообществе разработчиков ASP.NET MVC. Этот контейнер включает несколько специальных расширений для работы с ASP.NET и поддерживает простой в использовании текущий интерфейс для установки и регистрации зависимостей.

Для инициализации и работы с контейнером Ninject IoC понадобится настроить его начальный загрузчик. Этот начальный загрузчик отвечает за управление модулями, зарегистрированными с помощью Ninject. Важным модулем, с которым следует ознакомиться, является BindingsModule. Обратите внимание на статические методы Start() и Stop() начального загрузчика. Эти методы должны вызываться в методах запуска и завершения приложения внутри Global.asax. Ниже показано, как установить начальный загрузчик Ninject:

```
private static readonly Bootstrapper bootstrapper = new Bootstrapper();  
  
/// <summary>  
/// Запуск приложения.  
/// </summary>  
public static void Start()  
{  
    DynamicModuleUtility.RegisterModule(typeof(OnePerRequestModule));  
    DynamicModuleUtility.RegisterModule(typeof(  
        HttpApplicationInitializationModule));  
    bootstrapper.Initialize(CreateKernel);  
}  
  
/// <summary>  
/// Останов приложения.  
/// </summary>  
public static void Stop()  
{  
    bootstrapper.ShutDown();  
}  
  
/// <summary>  
/// Создание ядра, которое будет управлять приложением.  
/// </summary>  
/// <returns>Созданное ядро.</returns>  
private static IKernel CreateKernel()  
{  
    var kernel = new StandardKernel();  
    RegisterServices(kernel);  
    return kernel;  
}  
  
/// <summary>  
/// Загрузка модулей и регистрация служб.  
/// </summary>
```

```

    /// </summary>
    /// <param name="kernel">Ядро.</param>
    private static void RegisterServices(IKernel kernel)
    {
        kernel.Load(new BindingsModule());
    }

```

Класс BindingsModule унаследован от базового класса модулей Ninject и содержит регистрацию зависимостей, необходимую для ASP.NET MVC Framework. Обратите внимание, что зависимости контроллера и маршрута определены с использованием одиночной области действия. Это означает, что обе зависимости будут управляться как одиночный объект, и всегда будет создаваться только один экземпляр для каждой зависимости. Любые специальные зависимости должны быть зарегистрированы в методе Load():

```

public class BindingsModule : Ninject.Modules.NinjectModule
{
    public override void Load()
    {
        Bind<ControllerActions>()
            .ToMethod(x => ControllerActions.DiscoverControllerActions())
            .InSingletonScope();

        Bind<IRouteGenerator>().To<RouteGenerator>().InSingletonScope();

        Bind<DataContext>().ToSelf().InRequestScope()
            .OnDeactivation(x => x.SaveChanges());

        Bind< IRepository>().To<Repository>().InRequestScope()
            .WithConstructorArgument("isSharedContext", true);
    }
}

```

Во время регистрации собственных зависимостей можно определить, каким образом контейнер должен управлять их временем жизни, выбрать аргументы для передачи конструктору и определить поведение, выполняемое при деактивизации зависимости.

Использование инверсии управления для расширения ASP.NET MVC

Инфраструктура ASP.NET MVC Framework в значительной степени опирается на принцип инверсии управления. Инфраструктура содержит готовую стандартную фабрику контроллеров, которая поддерживает создание контроллера приложения за счет перехвата входного запроса, чтения его маршрута MVC и создания специфического контроллера, а затем вызова метода контроллера на основе определения маршрута. Другой крупной областью, где вступает в игру IoC, является управление механизмом представлений для приложения и контролирование потока выполнения между контроллером и соответствующим представлением (представлениями).

Реальная мощь IoC проявляется при расширении инфраструктуры ASP.NET Framework путем переопределения ее распознавателя зависимостей с применением собственной инверсии управления, чтобы получить непосредственный контроль над тем, как ASP.NET MVC управляет зависимостями и создает экземпляры объектов. Переопределение стандартного распознавателя зависимостей ASP.NET MVC сводится просто к реализации интерфейса `IDependencyResolver` и регистрации специального распознавателя зависимостей в ASP.NET MVC Framework.

Давайте для примера рассмотрим построение специального распознавателя зависимостей, который использует Ninject IoC. Первым делом, необходимо реализовать интерфейс `IDependencyResolver`, передав соответствующие вызовы экземпляру `IKernel` (класс контейнера IoC в Ninject):

```
public class CustomDependencyResolver : IDependencyResolver
{
    private readonly Ninject.IKernel _kernel;
    public CustomDependencyResolver(Ninject.IKernel kernel)
    {
        _kernel = kernel;
    }
    public object GetService(Type serviceType)
    {
        return _kernel.TryGet(serviceType);
    }
    public IEnumerable<object> GetServices(Type serviceType)
    {
        return _kernel.GetAll(serviceType);
    }
}
```

Затем следует зарегистрировать эту реализацию, вызвав статический метод `SetResolver()` класса `System.Web.Mvc.DependencyResolver`, как показано ниже:

```
Ninject.IKernel kernel = new Ninject.StandardKernel();
DependencyResolver.SetResolver(new CustomDependencyResolver(kernel));
```

Обратите внимание, что сначала создается экземпляр `IKernel` из Ninject для передачи в `CustomDependencyResolver`. Хотя каждый контейнер IoC реализуется по-своему, большинство инфраструктур IoC требуют конфигурирования контейнера перед тем, как можно будет создавать и распознавать зависимости. В этом примере класс `StandardKernel` из Ninject обеспечивает стандартную конфигурацию.

Однако приведенный выше фрагмент пока что работать не будет – потребуется сначала сообщить `StandardKernel` о классах и интерфейсах, которыми нужно управлять. В Ninject это делается с применением метода `Bind<T>()`. Например, следующий вызов сообщает Ninject о необходимости использования конкретной реализации `ErrorLogger` всякий раз, когда требуется `IErrorLogger`:

```
kernel.Bind<IErrorLogger>().To<ErrorLogger>();
```

Ниже показан пример добавления новой привязки для класса `ErrorLogger`:

```
// Регистрировать службы с контейнером.
kernel.Bind<IErrorLogger>().To<ErrorLogger>();
```

Принцип Don't Repeat Yourself

Don't Repeat Yourself (DRY), или “Не повторяйся” – это принцип проектирования, тесно связанный с принципами SOLID, который рекомендует разработчикам избегать дублирования одинакового или очень похожего кода.

Взгляните на приведенный ниже класс `SearchController`; видите ли вы в нем потенциальные нарушения принципа DRY? Все выглядит хорошо до тех пор, пока не принять во внимание то, что в приложении могут существовать десятки контроллеров, которые содержат в точности такой же или похожий код.

Поначалу может показаться удачной идеей перемещение метода CheckUserRight() в базовый класс контроллера, разделяемый всеми контроллерами. Такой подход будет работать, но ASP.NET MVC Framework предлагает даже лучший вариант: для поддержки этого поведения разработчик может создать специальный фильтр действия (ActionFilter).

```
public class SearchController : Controller
{
    public ActionResult Add(Product product)
    {
        if (CheckUserRights())
            // Код для добавления товара.
        return View();
    }

    public ActionResult Remove(Product product)
    {
        if (CheckUserRights())
            // Код для удаления товара.
        return View();
    }

    private bool CheckUserRights()
    {
        // Код для проверки, может ли пользователь выполнить эту операцию.
    }
}
```

Резюме

В этой главе рассматривались ключевые шаблоны и принципы проектирования, использованные во время разработки инфраструктуры ASP.NET MVC Framework. Те же самые принципы (разделение ответственности, инверсия управления и SOLID) разработчик может применять и для построения гибкого, удобного в сопровождении веб-приложения ASP.NET MVC. Использование этих принципов не сводится только к написанию лучшего кода. Они являются фундаментальными строительными блоками для разработки архитектуры веб-приложения.

Улучшение сайта с помощью AJAX

За прошедшие 20 лет концепция веб-приложения значительно изменилась. Первично для отображения текстового контента и ссылок на другие текстовые страницы в Интернете был предназначен язык HTML. Однако спустя некоторое время, пользователи и производители контента захотели от своих веб-страниц гораздо большего, поэтому многие веб-сайты начали использовать технологии JavaScript и Dynamic HTML, делая статический HTML-контент более интерактивным. В результате *AJAX* (Asynchronous JavaScript and XML – асинхронный JavaScript и XML) стал универсальным термином, которым обозначалась возможность выполнения асинхронных запросов к веб-серверу, избегая необходимости в переходе на новую страницу для получения актуальных данных.

Вместо передачи и перерисовки целой страницы, технология AJAX запрашивает контент асинхронно и затем применяет его для обновления различных разделов страницы. Технология AJAX обычно запрашивает контент одного из следующих двух типов: генерируемая сервером HTML-разметка, которую браузер внедряет напрямую в страницу, и низкоуровневые сериализированные данные, которые JavaScript-логика клиентской стороны использует для создания новой или обновления существующей HTML-разметки в браузере.

В этой главе будет показано, как задействовать мощные средства ASP.NET MVC, которые помогают встраивать приемы AJAX в разрабатываемые веб-приложения.

Частичная визуализация

Концепция выполнения HTTP-запроса к серверу и получение в ответ HTML-разметки является основой World Wide Web. Следовательно, выполнение еще одного запроса для дополнительной генерируемой сервером HTML-разметки, чтобы обновить или заменить раздел исходной страницы, кажется вполне логичным вариантом. Этот подход называется *частичной визуализацией*, и он является очень удобным вариантом для обеспечения простого и эффективного поведения AJAX. Технология частичной визуализации предусматривает выполнение асинхронного запроса к серверу, который отвечает порцией HTML-разметки, готовой к вставке прямо в текущую страницу.

Например, предположим, что имеется следующий документ ajax_content.html, который требуется асинхронно вставить в страницу:

```
<h2>This is the AJAX content!</h2>
```

А вот страница, в которую должен быть вставлен этот контент:

```
<html>
<body>
<h1>Partial Rendering Demo</h1>
<div id="container" />
</body>
</html>
```

В этом примере для пометки места, куда необходимо вставить динамический контент, используется элемент `<div id="container" />`. После этого элемент `<div id="container" />` можно заполнить контентом серверной стороны из `ajax_content.html`, применив метод `.load()` библиотеки jQuery:

```
$("#container").load('ajax_content.html')
```

Метод `.load()` осуществляет асинхронный запрос контента на стороне сервера, а затем внедряет его в элемент `#container`. После этого вызова модель DOM будет включать динамически извлеченный контент серверной стороны:

```
<html>
<body>
<h1>Partial Rendering Demo</h1>
<div id="container">
    <h2>This is the AJAX content!</h2>
</div>
</body>
</html>
```

Как показано в рассмотренном примере, подход частичной визуализации является простым и эффективным способом динамического обновления разделов веб-страниц. Более того, его невероятно легко реализовать в приложении ASP.NET MVC!

Визуализация частичных представлений

По большей части, ASP.NET MVC трактует запрос частичной визуализации точно так же, как любой другой запрос – запрос маршрутизируется на соответствующее действие контроллера, которое выполняет любую необходимую логику.

Отличие проявляется ближе к концу запроса, когда наступает время визуализации представления. При нормальном ходе действия может применяться вспомогательный метод `Controller.View()` для возврата `ViewResult`, тогда как в рассматриваемом случае вызывается вспомогательный метод `Controller.Partial()`, возвращающий `PartialViewResult`. Это очень похоже на `ViewResult`, за исключением того, что происходит визуализация только контента представления, но не его компоновки.

В целях демонстрации указанного отличия давайте сравним визуализированную разметку из частичного представления `Auction` и визуализированную разметку, сгенерированную “нормальным” представлением `Auction`.

Визуализация “нормального” представления

Следующее действие контроллера (`AuctionsController.cs`) вызывает уже знакомый вам вспомогательный метод `Controller.View()`:

```
public class AuctionsController : Controller
{
    public ActionResult Auction(long id)
    {
```

```
        var db = new DataContext();
        var auction = db.Auctions.Find(id);
        return View("Auction", auction);
    }
}
```

Соответствующее ему представление Auction (Auction.cshtml) выглядит следующим образом:

```
@model Auction


@Model.Title



![@Model.Title](@Model.ImageUrl)
        Current Price: <strong>@Model.CurrentPrice</strong>
        <span class="current-price">@Model.CurrentPrice</span>



### Description



@Model.Description


```

Приведенная комбинация производит финальную визуализированную HTML-разметку, показанную в примере 6.1.

Пример 6.1. Визуализированная разметка для “нормального” представления Auction

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <link href="/Content/Site.css" rel="stylesheet" type="text/css" />
    <link href="/Content/themes/base/jquery.ui.all.css" rel="stylesheet"
          type="text/css" />
    <script src="/Scripts/jquery-1.7.1.min.js"
           type="text/javascript"></script>
    <script src="/Scripts/jquery-ui-1.8.16.js"
           type="text/javascript"></script>
    <script src="/Scripts/modernizr-2.0.6.js" type="text/javascript">
    </script>
    <script src="/Scripts/AjaxLogin.js" type="text/javascript"></script>
</head>
<body>
<header>
    <h1 class="site-title"><a href="/">EBuy: The ASP.NET MVC Demo Site</a></h1>
    <nav>
        <ul id="menu">
            <li><a href="/categories/Electronics">Electronics</a></li>
            <li><a href="/categories/Home_and_Outdoors">Home/Outdoors</a></li>
            <li><a href="/categories/Collectibles">Collectibles</a></li>
        </ul>
    </nav>
</header>
<section id="main">
```

```

<div class="title">Xbox 360 Kinect Sensor with Game Bundle</div>
<div class="overview">
    
    <p>
        Closing in <span class="time-remaining">4 days, 19 hours</span>
    </p>
    <div>
        <a class="show-bid-history">
            href="/auctions/764cc5090c04/bids">Bid History</a>
    </div>
    <p>
        <strong>Current Price: </strong>
        <span class="current-price">$43.00</span>
    </p>
</div>
<h3>Description</h3>
<div class="description">
    You are the controller with Kinect for Xbox 360!
</div>
</section>
<footer>
    <p>&copy; 2012 - EBuy: The ASP.NET MVC Demo Site</p>
</footer>
</body>
</html>

```

Визуализация частичного представления

Обратите внимание, что представление `Auctions.cshtml` визуализируется внутри компоновки сайта. Это именно то, что пользователи должны видеть, когда они просматривают страницу в первый раз. Но что если необходимо повторно использовать разметку компоновки, которая в действительности остается той же самой для каждой страницы сайта, и обновлять только сведения об аукционном товаре, чтобы отображать детали другого аукционного товара, не требуя от пользователя перехода на другую страницу? Решение заключается в применении метода `Controller.PartialView()` для создания `PartialViewResult` вместо `ViewResult`, генерируемого методом `Controller.View()`:

```

public class AuctionsController : Controller
{
    public ActionResult Auction(long id)
    {
        var db = new DataContext();
        var auction = db.Auctions.Find(id);
        return View("Auction", auction);
    }
    public ActionResult PartialAuction(long id)
    {
        var db = new DataContext();
        var auction = db.Auctions.Find(id);
        return PartialView("Auction", auction);
    }
}

```

Как видите, ничего не изменилось кроме использования метода PartialView() вместо View(). Объект PartialViewResult может даже применять в точности те же самые представления, от которых зависит ViewResult. В действительности PartialViewResult и ViewResult почти идентичны за исключением одного очень важного отличия: PartialViewResult визуализирует *только* разметку самого представления, но никакой компоновки или мастер-страницы, которая может быть указана в представлении. Более того, частичные представления ведут себя аналогично нормальнym представлениям, поэтому вы можете использовать любой желаемый тип синтаксиса (например, Razor) и в полной мере задействовать функциональность представлений ASP.NET MVC, такую как вспомогательные методы HTML.



Поскольку частичные страницы не выполняют компоновку, может понадобиться включить некоторые зависимости, вроде CSS или JavaScript, прямо в частичное представление, а не в компоновку страницы.

Это означает, что для того же самого представления, показанного в Auction.cshtml в предыдущем разделе, результат PartialView будет визуализировать разметку, приведенную в примере 6.2.

Пример 6.2. Визуализированная разметка для частичного представления аукционного товара

```
<div class="title">Xbox 360 Kinect Sensor with Game Bundle</div>
<div class="overview">
    
    <p>
        Closing in <span class="time-remaining">4 days, 19 hours</span>
    </p>
    <div>
        <a class="show-bid-history"
            href="/auctions/764cc5090c04/bids">Bid History</a>
    </div>
    <p>
        <strong>Current Price: </strong>
        <span class="current-price">$43.00</span>
    </p>
</div>
<h3>Description</h3>
<div class="description">
    You are the controller with Kinect for Xbox 360!
</div>
```

При таких изменениях для загрузки HTML-разметки нового аукционного товара без необходимости в переходе на новую страницу можно применять следующий код jQuery клиентской стороны:

```
function showAuction(auctionId) {
    $('#main').load('/Auctions/PartialAuction/' + auctionId);
}
```



Если вы запишете предыдущий фрагмент внутри представления Razor, то сможете использовать класс `UrlHelper` инфраструктуры ASP.NET MVC для генерации корректного маршрута к действию `AuctionsController.Auction`.

Просто замените код:

```
'/Auctions/PartialAuction/' + auctionId
```

следующим кодом:

```
'@Url("PartialAuction", "Auctions")/' + auctionId
```

Управление сложностью с помощью частичных представлений

В предыдущем примере было показано, как использовать частичные представления для отображения страницы без ее компоновки. Однако частичные представления не всегда должны быть полными страницами – разделение страницы на множество частичных представлений часто является хорошим способом упрощения в противном случае чрезвычайно сложного представления.

Пожалуй, лучшим примером управления сложностью с помощью частичных представлений является ситуация, когда необходимо отобразить что-нибудь в цикле `foreach`, вроде списка аукционных товаров, как показано в следующем фрагменте:

```
@model IEnumerable<Auction>
<h1>Auctions</h1>
<section class="auctions">
@foreach(var auction in Model) {
    <section class="auction">
        <div class="title">@auction.Title</div>
        <div class="overview">
            
            <p>
                <strong>Current Price: </strong>
                <span class="current-price">@auction.CurrentPrice</span>
            </p>
        </div>
        <h3>Description</h3>
        <div class="description">
            @auction.Description
        </div>
    </section>
}
</section>
```

Внимательно взгляните на разметку внутри цикла `foreach` – напоминает ли вам она что-либо виденное ранее? Это *абсолютно такая же разметка из Auctions.cshtml*, которая применялась в качестве результата частичной визуализации!

Это означает, что должна быть возможность замены полного раздела вызовом вспомогательного метода `Html.Partial()` для визуализации уже созданного частичного представления:

```
@model IEnumerable<Auction>
<h1>Auctions</h1>
<section class="auctions">
```

```
@foreach(var auction in Model) {  
    <section class="auction">  
        @Html.Partial("Auction", auction)  
    </section>  
}  
</section>
```

Обратите внимание на возможность указания модели, которую визуализирует частичное представление, передав ее во втором параметре вспомогательному методу `Html.Partial()`. Это позволяет пройти в цикле по всему списку объектов `Auction` и применить одно и то же представление к каждому экземпляру.

Как демонстрируют примеры в этом разделе, эффективное применение частичных представлений не только помогает упростить каждое индивидуальное представление и сократить объем дублированного кода в приложении, но также позволяет поддерживать согласованность в рамках всего сайта.

Визуализация с помощью JavaScript

Хотя подход с предварительно визуализированной HTML-разметкой является очень простым и эффективным, может быть весьма неэкономно передавать и данные, которые необходимо отобразить, и разметку, предназначенную для их отображения, в ситуации, когда браузер вполне способен создать такую разметку самостоятельно. Таким образом, альтернатива извлечению предварительно визуализированной HTML-разметки заключается в извлечении низкоуровневых данных для отображения и затем их использование для создания и обновления HTML-элементов за счет манипулирования моделью DOM напрямую.

Чтобы реализовать подход с визуализацией на стороне клиента, потребуется наличие двух вещей: сервера, который может произвести сериализованные данные, и логики клиентской стороны, которой известно, как разобрать эти сериализованные данные и преобразовать их в HTML-разметку.

Визуализация данных JSON

Давайте примемся сначала за серверную часть: ответ на запрос AJAX выдачей сериализованных данных. Однако перед тем как это можно будет сделать, понадобится принять решение относительно применяемой технологии сериализации данных.

Нотация объектов JavaScript (JavaScript Object Notation – JSON) – это простой и очень эффективный формат для передачи данных через веб. Для представления данных объект JSON использует два типа структур данных: коллекции пар “имя/значение” и упорядоченные списки значений (т.е. массивы). Как должно быть понятно по названию, формат JavaScript Object Notation основывается на подмножестве языка JavaScript, поэтому все современные браузеры должны понимать его.

Инфраструктура ASP.NET MVC предлагает встроенную поддержку JSON в форме результата действия `JsonResult`, который принимает объект модели и сериализирует его в формат JSON. Чтобы добавить к своим действиям контроллеров поддержку AJAX через JSON, просто воспользуйтесь методом `Controller.Json()` для создания нового экземпляра `JsonResult`, содержащего предназначенный для сериализации объект.

Для демонстрации работы вспомогательного метода `Json()` и `JsonResult` давайте добавим к контроллеру `AuctionsController` действие `JsonAuction`:

```

public ActionResult JsonAuction(long id)
{
    var db = new DataContext();
    var auction = db.Auctions.Find(id);
    return Json(auction, JsonRequestBehavior.AllowGet);
}

```

Это новое действие контроллера отвечает на запросы выдачей сериализованной в формате JSON версии данных auction. Например:

```

{
    "Title": "XBOX 360",
    "Description": "Brand new XBOX 360 console",
    "StartTime": "01/12/2012 12:00 AM",
    "EndTime": "01/31/2012 12:00 AM",
    "CurrentPrice": "$199.99",
    "Bids": [
        {
            "Amount" : "$200.00",
            "Timestamp": "01/12/2012 6:00 AM"
        },
        {
            "Amount" : "$205.00",
            "Timestamp": "01/14/2012 8:00 AM"
        },
        {
            "Amount" : "$210.00",
            "Timestamp": "01/15/2012 12:32 PM"
        }
    ]
}

```

Приведенный фрагмент является великолепным примером простоты и элегантности JSON – обратите внимание, что скобки [и] определяют массивы, а значениями свойств могут быть простые или сложные типы данных, представленные самими объектами JSON. Этот синтаксис также дает положительный побочный эффект, заключающийся в том, что данные JSON становятся удобными для человеческого восприятия.

Предотвращение уязвимости JSON Hijacking с помощью перечисления JsonRequestBehavior

Обратите внимание, что в качестве второго параметра метода Json() указано значение перечисления JsonRequestBehavior.AllowGet, которое явно информирует инфраструктуру ASP.NET MVC Framework о том, что в ответ на HTTP-запрос GET разрешено возвращать данные JSON. Параметр JsonRequestBehavior.AllowGet в этом случае необходим, поскольку по умолчанию ASP.NET MVC запрещает возврат данных JSON в ответ на HTTP-запрос GET, чтобы предотвратить потенциально опасную уязвимость, которая известна под названием *JSON Hijacking* (похищение JSON). Эта уязвимость возникает из-за проблемы в способе обработки многими браузерами JavaScript-дескрипторов <script>, который может привести к раскрытию конфиденциальной информации, если данные в запросе включают массив JSON.

Несмотря на определенную сложность, в действительности для предотвращения этой уязвимости никогда не следует возвращать в запросе GET данные, которые не

должны быть видны внешнему миру. Таким образом, ASP.NET MVC заставляет взвешенно подходить к доставке данных JSON подобным незащищенным способом при возврате публично доступных (не конфиденциальных) данных, явно указывая значение перечисления `JsonRequestBehavior.AllowGet`. В сценариях, при которых необходимо передавать конфиденциальную информацию через запрос JSON, от этой уязвимости можно защититься, ограничивая доступ к методу контроллера HTTP-запросами POST с применением атрибута `HttpPostAttribute`:

```
[HttpPost]
public ActionResult JsonAuction(long id)
{
    var db = new DataContext();
    var auction = db.Auctions.Find(id);
    return Json(auction);
}
```

Запрашивание данных JSON

Имея функциональность серверной стороны, теперь необходимо выполнить запрос для извлечения данных JSON, чтобы их можно было использовать для построения разметки на стороне клиента. К счастью, jQuery позволяет делать это очень легко.

Чтобы запросить данные JSON из действия контроллера ASP.NET MVC, просто добавьте обращение `$.ajax()` к URL действия контроллера и укажите функцию `success` для обработки ответа. Первый параметр функции `success` (имеющий имя `result` в приведенном ниже примере) содержит десериализованный объект, возвращенный из сервера.

Сказанное демонстрируется в следующем фрагменте на вызове нового действия контроллера `JsonAuction` и применении методов `.val()` и `.html()` библиотеки jQuery для обновления модели DOM сериализованными в формате JSON данными `Auction`, которые возвратило действие контроллера:

```
function updateAuctionInfo(auctionId) {
    $.ajax({
        url: "/Auctions/JsonAuction/" + auctionId,
        success: function (result) {
            $('#Title').val(result.Title);
            $('#Description').val(result.Description);
            $('#CurrentPrice').html(result.CurrentPrice);
        }
    });
}
```

Хотя этот подход клиентской стороны может потребовать несколько большего объема кода, он позволяет отправлять по сети наименьшее количество данных, что в целом делает его намного более эффективным методом передачи и отображения данных AJAX.

Шаблоны клиентской стороны

Показанный выше прием с конкатенацией строк является удобным способом генерации разметки клиентской стороны, однако он подходит только для небольших разделов разметки. По мере роста объема разметки увеличивается и сложность кода, необходимого для конкатенации, давая в результате код, который все сложнее и сложнее сопровождать.

Шаблоны клиентской стороны – это мощная альтернатива простой конкатенации строк, которая позволяет быстро и эффективно трансформировать данные JSON в HTML-разметку легко сопровождаемым способом. Шаблоны клиентской стороны определяют многократно используемые разделы разметки путем комбинирования простой HTML-разметки с выражениями над данными, которые варьируются от простых заполнителей, заменяемых значениями данных, до полноценной логики JavaScript, осуществляющей обработку данных прямо внутри шаблона.

Следует отметить, что концепция шаблонов клиентской стороны не является частью какой-то официальной спецификации, поэтому для реализации этого подхода необходимо полагаться на некоторую библиотеку JavaScript. Хотя точный синтаксис в разных библиотеках будет отличаться, фундаментальная концепция остается той же самой: библиотека получает разметку клиентского шаблона и разбирает ее в функцию, которой известно, как произвести HTML-разметку из объекта JSON.

В приведенных ниже примерах используется синтаксис шаблонов Mustache (<http://mustache.github.com/>) для определения разметки клиентского шаблона и JavaScript-библиотека mustache.js (<https://github.com/janl/mustache.js>) для разбора и выполнения клиентских шаблонов в браузере. Тем не менее, на выбор доступно множество библиотек клиентских шаблонов, так что исследуйте их и выясните, какая из них лучше всего удовлетворяет нуждам текущего приложения.

Для демонстрации работы шаблонов клиентской стороны преобразуем представление Auction, показанное ранее в главе, в синтаксис клиентских шаблонов:

```
<div class="title">{{Title}}</div>
<div class="overview">
  
  <p>
    <strong>Current Price: </strong>
    <span class="current-bid">{{CurrentPrice}}</span>
  </p>
</div>
<h3>Description</h3>
<div class="description">
  {{Description}}
</div>
```

Обратите внимание, что разметка клиентского шаблона выглядит в основном идентично финальному выводу. На самом деле единственное отличие заключается в том, что клиентский шаблон содержит заполнители данных вместо действительных данных. Вдобавок следует отметить, что этот очень простой пример призван продемонстрировать фундаментальную концепцию шаблонов клиентской стороны – большинство библиотек клиентских шаблонов предлагают *намного больше* функциональности, нежели простые заполнители.

Второй шаг состоит в *компиляции* клиентского шаблона, или преобразовании HTML-разметки клиентского шаблона в исполняемую JavaScript-функцию.



Поскольку компиляция шаблона обычно представляет собой наиболее дорогостоящую операцию в рамках процесса, часто удобно скомпилировать функцию заранее и сохранить ее в переменной.

В этом случае можно будет запускать скомпилированный шаблон много раз, выполнив саму компиляцию только однажды.

Наконец, мы вызываем скомпилированный шаблон, передавая ему данные, которые необходимо преобразовать в HTML-разметку. Затем скомпилированный шаблон возвращает форматированную HTML-разметку, которую можно вставить в произвольное место DOM.

В примере 6.3 приведена полная страница, использующая клиентские шаблоны.

Пример 6.3. Завершенная страница, в которой применяются клиентские шаблоны

```
@model IEnumerable<Auction>
<h2>Auctions</h2>
<ul class="auctions">
    @foreach(var auction in Model) {
        <li class="auction" data-key="@auction.Key">
            <a href="#">
                
                <span>@auction.Title</span>
            </a>
        </li>
    }
</ul>
<section id="auction-details">
    @Html.Partial("Auction", Model.First())
</section>
<script id="auction-template" type="text/x-template">
    <div class="title">{{Title}}</div>
    <div class="overview">
        
        <p>
            <strong>Current Price: </strong>
            <span class="current-bid">{{CurrentPrice}}</span>
        </p>
    </div>
    <h3>Description</h3>
    <div class="description">
        {{Description}}
    </div>
</script>
<script type="text/javascript" src="~/scripts/mustache.js"></script>
<script type="text/javascript">
    $(function() {
        var templateSource = $('#auction-template').html();
        var template = Mustache.compile(templateSource);
        $('.auction').click(function() {
            var auctionId = $(this).data("key");
            $.ajax({
                url: '@Url.Action("JsonAuction", "Auctions")/' + auctionId,
                success: function(auction) {
                    var html = template(auction);
                    $('#auction-details').html(html);
                }
            });
        });
    });
</script>
```

Хотя может показаться, что в этом примере много чего происходит, на самом деле код довольно прост.

1. Когда страница загружается, блок сценария в нижней части извлекает разметку клиентского шаблона из внутреннего HTML-контента элемента `auction-template`.
2. После этого сценарий передает разметку клиентского шаблона методу `Mustache.compile()` для ее компиляции в JavaScript-функцию, которая сохраняется в переменной `template`.
3. Затем блок сценария прослушивает событие щелчка на каждом элементе `Auction` в списке и при его возникновении инициирует запрос AJAX для извлечения сериализированных в формате JSON данных, связанных с элементом `Auction`, на котором был совершен щелчок.
 - В случае успешного извлечения данных `Auction` обработчик `success` выполняет ранее скомпилированный клиентский шаблон (хранящийся в переменной `template`) для генерации разметки на основе данных JSON.
 - Наконец, обработчик `success` вызывает метод `.html()` для замены контента элемента `auction-details` разметкой, сгенерированной в функции `template`.



MIME-тип "text/x-template" – это произвольный искусственный тип; здесь можно использовать практически любое недопустимое значение для MIME-типа.

Браузеры игнорируют дескрипторы `<script>`, в которых указаны MIME-типы, не распознаваемые браузерами, поэтому помещение разметки шаблона в дескриптор `<script>` и установка его MIME-типа в "недопустимое" значение, такое как "text/x-template", предотвращает визуализацию шаблона как обычной HTML-разметки вместе с остальной частью страницы.

Несмотря на то что подход с клиентскими шаблонами может выглядеть трудоемким, в большинстве случаев простота сопровождения и низкая полоса пропускания вполне окупают первоначальные затраты. Если в приложении используется множество AJAX-взаимодействий, которые дают в результате сложную разметку на стороне клиента, то клиентские шаблоны часто являются удачным выбором.

Повторное использование логики в запросах AJAX и не AJAX

Шаблон "модель-представление-контроллер", управляющий инфраструктурой ASP.NET MVC Framework, использует строгое разделение ответственности для обеспечения изоляции индивидуальных компонентов друг от друга. Хотя действия контроллера `PartialAuction` и `JsonAuction` могут делать в точности то, что необходимо, если уделить время на тщательное их исследование, выяснится, что было нарушено несколько шаблонов и рекомендаций, которые являются довольно фундаментальными в рамках философии MVC.

Когда все сделано правильно, логика приложения MVC не должна быть привязана к конкретному представлению. Тогда почему мы имеем *три* действия контроллера (показанный в примере 6.4), которые выполняют ту же самую логику, отличаясь только способом возврата контента браузеру?

Пример 6.4. AuctionsController.cs с тремя способами извлечения объекта Auction

```
public class AuctionsController : Controller
{
    public ActionResult Auction(long id)
    {
        var db = new DataContext();
        var auction = db.Auctions.Find(id);
        return View("Auction", auction);
    }

    [HttpPost]
    public ActionResult JsonAuction(long id)
    {
        var db = new DataContext();
        var auction = db.Auctions.Find(id);
        return Json(auction);
    }

    public ActionResult PartialAuction(long id)
    {
        var db = new DataContext();
        var auction = db.Auctions.Find(id);
        return PartialView("Auction", auction);
    }
}
```

Реагирование на запросы AJAX

Чтобы помочь справиться с таким дублированием логики и кода, инфраструктура ASP.NET MVC предоставляет расширяющий метод `Request.IsAjaxRequest()`, который позволяет выяснить, является ли запрос запросом AJAX. Затем эту информацию можно использовать для определения формата, в котором ожидает получить ответ запрашивающая сторона, и результата действия, который должен быть выбран для генерации этого ответа.



Метод `Request.IsAjaxRequest()` довольно прост: он лишь проверяет HTTP-заголовки входящего запроса на предмет того, что значением заголовка `X-Requested-With` является `XMLHttpRequest`, которое автоматически добавляется большинством браузеров и инфраструктур AJAX.

Если когда-либо понадобится заставить инфраструктуру ASP.NET MVC считать какой-нибудь запрос AJAX-запросом, просто добавьте HTTP-заголовок `Requested-With: XMLHttpRequest`.

В целях демонстрации функционирования метода `Request.IsAjaxRequest()` давайте сначала попробуем объединить вместе действия контроллера `Auction` и `PartialAuction`. Комбинированное действие контроллера должно извлечь экземпляр аукционного товара из контекста данных, а затем выбрать способ его отображения. Если это запрос AJAX, будет использоваться метод `PartialView()` для возврата `PartialViewResult`, а в противном случае — метод `View()` для возврата `ViewResult`:

```
public ActionResult Auction(long id)
{
    var db = new DataContext();
    var auction = db.Auctions.Find(id);
    if (Request.IsAjaxRequest())
        return PartialView("Auction", auction);
    return View("Auction", auction);
}
```

После такого изменения действие контроллера `Auction` способно отвечать как на "нормальные" HTTP-запросы GET, так и на запросы AJAX выдачей соответствующего представления, используя ту же самую логику приложения.

Реагирование на запросы JSON

К сожалению, ASP.NET не предоставляет удобный метод вроде `Request.IsAjaxRequest()`, который помог бы выяснить, ожидаются ли на стороне запроса данные JSON. Однако, проявив небольшую смекалку, эту логику довольно легко реализовать самостоятельно. Мы начнем с простейшего решения: добавление к действию контроллера специального параметра, который указывает на то, что запрос ожидает внутри ответа данные JSON.

Например, можно просматривать параметр запроса по имени `format` и возвращать `JsonResult`, когда значением этого параметра является "json":

```
public ActionResult Auction(long id)
{
    var db = new DataContext();
    var auction = db.Auctions.Find(id);
    if (string.Equals(request["format"], "json"))
        return Json(auction);
    return View("Auction", auction);
}
```

Затем клиенты могут запрашивать данные об аукционном товаре из этого действия в формате JSON, добавляя в конец строки запроса конструкцию `?format=json`; к примеру, `/Auctions/Auction/1234?format=json`.

Можно также пойти еще дальше и перенести эту логику в собственный расширяющий метод, чтобы можно было вызывать его откуда угодно, почти как расширяющий метод `Request.IsAjaxRequest()`:

```
using System;
using System.Web;

public static class JsonRequestExtensions
{
    public static bool IsJsonRequest(this HttpRequestBase request)
    {
        return string.Equals(request["format"], "json");
    }
}
```

Имея расширяющий метод `IsJsonRequest()`, предыдущий фрагмент можно переписать следующим образом:

```

public ActionResult Auction(long id)
{
    var db = new DataContext();
    var auction = db.Auctions.Find(id);

    if (Request.IsJsonRequest())
        return Json(auction);

    return View("Auction", auction);
}

```

Применение одной и той же логики во множестве действий контроллера

Если скомбинировать частичную визуализацию и показанные выше условные подходы JSON в одном действии контроллера, мы получим очень гибкий подход, который способен производить разный вывод на основе одной и той же логики приложения. Взгляните на оптимизированный контроллер `AuctionsController.cs`:

```

public class AuctionsController : Controller
{
    public ActionResult Auction(long id)
    {
        var db = new DataContext();
        var auction = db.Auctions.Find(id);

        // Реагировать на запросы AJAX.
        if (Request.IsAjaxRequest())
            return PartialView("Auction", auction);

        // Реагировать на запросы JSON.
        if (Request.IsJsonRequest())
            return Json(auction);

        // Установить по умолчанию "нормальное" представление с компоновкой.
        return View("Auction", auction);
    }
}

```

Код, управляющий этим действием контроллера, может быть гибким, но тот факт, что он определен внутри действия контроллера `Auction`, означает, что его не могут использовать никакие другие действия.

К счастью, ASP.NET MVC предлагает великолепный механизм повторного использования логики во множестве действий контроллера — *фильтры действий*.

Чтобы переместить эту логику в фильтр действия, который может быть применен к другим действиям контроллера, начать следует с создания класса, реализующего тип `System.Web.Mvc.ActionFilterAttribute` и переопределяющего метод `OnActionExecuted()` данного типа. Это позволит модифицировать результат действия после того, как оно выполнено, но перед выполнением самого результата действия:

```

public class MultipleResponseFormatsAttribute : ActionFilterAttribute
{
    public override void OnActionExecuted(ActionExecutedContext filterContext)
    {
        // Мы поместим сюда необходимую логику.
    }
}

```

Затем можно переместить логику из действия контроллера Auction в созданный новый класс и применять его для замены результата действия, который контроллер возвращал первоначально (`filterContext.Result`), когда текущим является запрос AJAX или JSON:

```
using System;
using System.Web.Mvc;

public class MultipleResponseFormatsAttribute : ActionFilterAttribute
{
    public override void OnActionExecuted(ActionExecutedContext filterContext)
    {
        var request = filterContext.HttpContext.Request;
        var viewResult = filterContext.Result as ViewResult;
        if (viewResult == null)
            return;
        if (request.IsAjaxRequest())
        {
            // Заменить результат экземпляром PartialViewResult.
            filterContext.Result = new PartialViewResult
            {
                TempData = viewResult.TempData,
                ViewData = viewResult.ViewData,
                ViewName = viewResult.ViewName,
            };
        }
        else if (Request.IsJsonRequest())
        {
            // Заменить результат экземпляром JsonResult.
            filterContext.Result = new JsonResult
            {
                Data = viewResult.Model
            };
        }
    }
}
```

Теперь можно легко применить фильтр действия `MultipleResponseFormats Attribute` к любому действию контроллера внутри веб-сайта, незамедлительно добавив возможность динамического выбора между возвратом представления, частичного представления или ответа JSON в зависимости от входящего запроса.

Отправка данных на сервер

Первая половина этой главы была посвящена запрашиванию контента и данных от сервера через AJAX. А теперь следует посмотреть на вторую часть уравнения: отправка данных *на* сервер посредством AJAX.

Двумя наиболее популярными способами “отправки” данных на любой веб-сервер являются параметры строки запроса (обычно через HTTP-запрос GET) и данные отправки формы (обычно через HTTP-запрос POST). В главе 1 этой книги была введена концепция привязки моделей ASP.NET MVC, демонстрирующая возможность ASP.NET MVC “автоматического” заполнения параметров действия контроллера значениями из запроса.

Однако в главе 1 не было показано, что в дополнение к отображению на значения строки запроса и “нормальной” отправки HTTP-формы, инфраструктуре привязки моделей ASP.NET MVC также известно, как осуществлять привязку объектов JSON к параметрам действия. Это означает, что для приема данных JSON действиями контроллера вообще ничего не придется делать – все уже работает!

Таким образом, теперь мы переходим к стороне клиента, где jQuery-метод `$.post()` существенно упрощает отправку данных JSON действиям контроллера. Чтобы отправить объект на сервер, нужно просто предоставить желаемый URL для отправки и объект, содержащий то, что необходимо отправить – jQuery позаботится о сериализации объекта в JSON и присоединении его в виде данных отправки формы к запросу.

Давайте рассмотрим пример, в котором новый объект `Auction` заполняется с помощью JavaScript и отправляется действию контроллера `Create` (как показано в следующем примере):

```
var auction = {  
    "Title": "New Auction",  
    "Description": "This is an awesome item!",  
    "StartPrice": "$5.00",  
    "StartTime": "01/12/2012 6:00 AM",  
    "EndTime": "01/19/2012 6:00 AM"  
};  
$.post('@Url.Action("Create", "Auctions")', auction);
```

Вспомните, что действие контроллера не должно делать что-то особенное – данные JSON автоматически привязываются к параметру `auction` действия контроллера. Действию контроллера остается лишь выполнить его логику (т.е. добавить аукционный товар к базе данных) и возвратить результат. Действие контроллера `Create` выглядит примерно так:

```
[HttpPost]  
public ActionResult Create(Auction auction)  
{  
    if (ModelState.IsValid)  
    {  
        var db = new DataContext();  
        db.Auctions.Add(auction);  
        return View("Auction", auction);  
    }  
    return View("Create", auction);  
}
```

Отправка сложных объектов JSON

Стандартная логика привязки моделей JSON обладает одним важным ограничением: она реализует подход “все или ничего”. Это значит, что фабрика ожидает *полный* ответ для включения одиночного объекта JSON и не поддерживает возможность индивидуальных полей доставлять данные в формате JSON.

Рассмотрим пример, в котором стандартный подход приведет к проблемам. Предположим, что имеется коллекция объектов `Bid`, в каждом из которых определены два свойства, `Amount` и `Timestamp`:

```
Bid[0].Timestamp="01/12/2012 6:00 AM" &  
Bid[0].Amount=100.00 &  
Bid[1].Timestamp="01/12/2012 6:42 AM" &  
Bid[1].Amount=73.64
```

Ниже показано, как тот же самый запрос будет представлен в формате JSON:

```
[  
    { "Timestamp":"01/12/2012 6:00 AM", "Amount":100.00 },  
    { "Timestamp":"01/12/2012 6:42 AM", "Amount":73.64 }  
]
```

Массив JSON не только существенно понятнее, проще и меньше, его также намного легче строить и управлять с использованием JavaScript в браузере. Эта простота особенно полезна при динамическом построении формы, позволяя пользователю предлагать свою цену на множество аукционов одновременно.

Тем не менее, для представления поля Bid в формате JSON с применением DefaultModelBinder понадобится отправить весь объект в виде JSON. Например:

```
{  
    Bids:  
    [  
        { "Timestamp":"01/12/2012 6:00 AM", "Amount":100.00 },  
        { "Timestamp":"01/12/2012 6:42 AM", "Amount":73.64 }  
    ],  
}
```

На первый взгляд, отправка объектов JSON, предназначенных для использования в привязке модели, выглядит неплохой идеей. Тем не менее, этот подход обладает рядом недостатков.

Во-первых, клиент должен строить полный запрос динамически, и эта логика должна знать, как явно управлять каждым полем, которое может быть отправлено; HTML-форма перестает быть формой и становится способом сбора JavaScript-логикой данных от пользователя. Во-вторых, на стороне пользователя поставщик значений JSON игнорирует все HTTP-запросы за исключением тех, в которых заголовок Content-Type установлен в "application/json", поэтому такой подход не будет работать для стандартных запросов GET; он функционирует только для запросов AJAX, содержащих корректный заголовок. Наконец, в-третьих, если стандартная логика проверки достоверности дает отказ на всего лишь одном поле, связыватель модели считает недопустимым *целый объект*!

Чтобы уменьшить влияние этих недостатков, мы можем ввести альтернативу встроенной логике привязки моделей JSON, создав специальный связыватель модели JSON, показанный в примере 6.5. Класс JsonModelBinder отличается от фабрики поставщика значений JSON в том, что он позволяет каждомуциальному полю содержать JSON, устранив необходимость в отправке целого запроса как одиночного объекта JSON. Поскольку связыватель модели привязывает каждое свойство по отдельности, можно устанавливать, какие поля содержать простые значения, а какие поддерживают данные JSON. Как и большинство специальных связывателей моделей, связыватель модели JSON является производным от DefaultModelBinder, так что он может вернуться к стандартной логике привязки, когда поля не содержат данные JSON.

Пример 6.5. Класс JsonModelBinder

```
public class JsonModelBinder : DefaultModelBinder  
{  
    public override object BindModel  
    (  
        ControllerContext controllerContext,  
        ModelBindingContext bindingContext  
    )
```

```

{
    string json = string.Empty;
    var provider = bindingContext.ValueProvider;
    var providerValue = provider.GetValue(bindingContext.ModelName);
    if (providerValue != null)
        json = providerValue.AttemptedValue;
    // Базовое выражение, которое проверяет, имеет ли строка в начале и конце
    // символы объекта JSON ( {} ) или массива ( [] ).
    if (Regex.IsMatch(json, @"^(\[.*\]|{.*})$"))
    {
        return new JavaScriptSerializer()
            .Deserialize(json, bindingContext.ModelType);
    }
    return base.BindModel(controllerContext, bindingContext);
}
}

```

Выбор связывателя модели

Учитывая ориентацию на расширяемость ASP.NET MVC, вряд ли вызовет удивление тот факт, что в инфраструктуре имеется довольно много способов указания того, какой связыватель модели должен использоваться для любой заданной модели. На самом деле комментарии в исходном коде метода `ModelBinderDictionary.GetBinder()` объясняют, каким образом инфраструктура обнаруживает подходящий связыватель модели для каждого типа:

```

private IModelBinder GetBinder(Type modelType, IModelBinder fallbackBinder)
{
    // Попытаться найти связыватель для этого типа.
    // Используется следующий порядок приоритетов.
    // 1. Связыватель, возвращенный из поставщика.
    // 2. Связыватель, зарегистрированный в глобальной таблице.
    // 3. Атрибут связывателя, определенный для типа.
    // 4. Поставляемый резервный связыватель.
}

```

Давайте рассмотрим этот список снизу вверх.

Замена стандартного (резервного) связывателя

В отсутствие дополнительной конфигурации ASP.NET MVC будет привязывать все модели с использованием `DefaultModelBinder`. Этот глобальный стандартный обработчик можно заменить, указав в свойстве `ModelBinders.Binders.DefaultBinder` новый связыватель модели. Например:

```

protected void Application_Start()
{
    ModelBinders.Binders.DefaultBinder = new JsonModelBinder();
    // ...
}

```

После такой установки экземпляр `JsonModelBinder` будет действовать в качестве нового резервного связывателя, обрабатывая привязку всех моделей, если не указано иное.

Оснащение моделей специальными атрибутами

Пожалуй, наиболее элегантным подходом к указанию связывателей моделей является использование абстрактного класса `System.Web.Mvc.CustomModelBinderAttribute` для декорирования классов и отдельных свойств удобным декларативным путем. Хотя этот подход можно применить к любой модели, которую необходимо привязать, он лучше всего сочетается с подходом модели запроса, т.к. привязка моделей представляет собой единственную причину существования модели запроса!

Чтобы можно было использовать подход `CustomModelBinderAttribute`, сначала понадобится создать реализацию. Ниже приведен пример реализации абстрактного класса `CustomModelBinderAttribute` и его применения к модели `CreateProductRequest`:

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Enum |
                AttributeTargets.Interface | AttributeTargets.Parameter |
                AttributeTargets.Struct | AttributeTargets.Property,
                AllowMultiple = false, Inherited = false)]
public class JsonModelBinderAttribute : CustomModelBinderAttribute
{
    public override IModelBinder GetBinder()
    {
        return new JsonModelBinder();
    }
}

public class CreateProductRequest
{
    // ...

    [Required]
    [JsonModelBinder]
    public IEnumerable<CurrencyRequest> UnitPrice { get; set; }
}
```

Декорирование `CreateProductRequest.UnitPrice` атрибутом `JsonModelBinderAttribute` указывает, что связыватель модели должен использовать `JsonModelBinder` (созданный с помощью вызова `JsonModelBinderAttribute.GetBinder()`) для привязки свойства `CreateProductRequest.UnitPrice`.

Это будет работать, если только для типа `CurrencyRequest` не был зарегистрирован глобальный обработчик или поставщик связывателей модели.

Регистрация глобального связывателя

Во многом тем же способом, которым можно устанавливать стандартный связыватель модели в качестве резервного для всех типов моделей, имеется также возможность регистрации связывателей моделей для отдельных типов. Как и в случае установки стандартного связывателя модели, синтаксис очень прост.

Следующий пример сообщает инфраструктуре о необходимости применения `JsonModelBinder` для каждой встретившейся модели `Currency`:

```
ModelBinders.Binders.Add(typeof(Currency), new JsonModelBinder());
```

Такой подход позволяет ассоциировать связыватель модели с конкретным типом в рамках целого приложения в единственной строке. Он также является эффективным способом управления привязкой бизнес-моделей без необходимости в декорировании этих моделей специальными атрибутами связывателей модели.

Эффективная отправка и получение данных JSON

Формат JSON является фундаментальным строительным блоком для создания насыщенных интерактивных веб-приложений на основе AJAX, поэтому важно понимать, как правильно его использовать. В следующем разделе будет показано, что jQuery упрощает работу с данными JSON и элементами HTML.

Одним из наиболее сложных аспектов, с которыми приходится иметь дело при работе с данными JSON, является сериализация. Проблемы могут возникать со сложными объектами, которые имеют множество отношений или тесно связаны со специфичной технологией доступа к данным, такой как Entity Framework. Если при возврате результата JSON переданный объект не может быть сериализован, возвращается внутренняя ошибка сервера с кодом 500.

Другой существенный недостаток работы со сложными объектами заключается в том, что они могут создавать трудности при обработке посредством JavaScript. Хорошая практика, помогающая избежать этих проблем, предусматривает создание специальной облегченной версии сущности под названием *объект передачи данных* (data transfer object – DTO), который более просто преобразуется в JSON. Объекты DTO должны использовать простые структуры данных и избегать сложных многоуровневых отношений.

Вдобавок объекты DTO должны содержать только поля, которые требуются приложению или запросу, и не более того. Вполне допустимо иметь множество классов DTO – даже для одной и той же сущности – выступающих в качестве ответов для различных запросов. Ниже приведен пример простого объекта DTO. Его упрощенная структура данных специально предназначена для работы с JavaScript. Более того, тот факт, что объект DTO меньше, чем модель Auction, делает его оптимальной структурой данных для ответа AJAX:

```
public class AuctionDto
{
    public string Title { get; set; }
    public string Description { get; set; }
}
```

Междоменный AJAX

По умолчанию веб-браузеры ограничивают вызовы AJAX единственным источником, которым является веб-приложение. Это ограничение призвано предотвращать проблемы безопасности, подобные атакам межсайтовыми сценариями (XSS). Тем не менее, иногда приложениям требуется возможность взаимодействия с внешне размещаемыми API-интерфейсами REST (Representational State Transfer – передача состояния представления) вроде Twitter или Google.

Чтобы такие сценарии работали, внешне размещаемые веб-приложения должны поддерживать запросы JSONP или разделение ресурсов между источниками (Cross-Origin Resource Sharing – CORS). Инфраструктура ASP.NET MVC не предлагает прямой поддержки этих опций; добавление указанных возможностей требует некоторого кодирования и конфигурирования.

JSONP

JSONP (JSON with Padding – JSON с дополнением; JSON с набивкой) – это искусственный трюк, который использует средство Cross-Site Request Forgery (подделка межсайтовых

запросов), описанное в разделе “Подделка межсайтовых запросов” главы 9, позволяя осуществлять междоменные вызовы AJAX, даже когда браузеры пытаются всячески препятствовать этому.

На высоком уровне взаимодействие JSONP сводится к следующим шагам.

1. Клиент создает функцию JavaScript, которая должна вызываться при получении ответа JSONP от сервера; например, `updateAuction`.
2. Клиент динамически добавляет дескриптор `<script>` в модель DOM, заставляя браузер считать, что он выполняет стандартное включение сценария, и пользуясь тем фактом, что браузеры разрешают в `<script>` ссылаться на другие домены.
3. Дескриптор `<script>` ссылается на вызов службы данных, которая поддерживает JSONP, и клиент указывает имя функции обратного вызова, созданной на шаге 1, в URL; например, `<script href="http://other.com/auctions/1234?callback=updateAuction" />`.
4. Сервер обрабатывает запрос и переходит к его визуализации, как это происходит с любым другим запросом JSON, но с одним важным отличием: вместо возвращения объекта JSON в качестве полного содержимого ответа сервер помещает этот объект внутрь обращения к функции *обратного вызова* клиентской стороны, имя которой было предоставлено клиентом (как показано в примере ниже).

Обратите внимание, что сервер не знает и не заботится о том, что делает эта функция обратного вызова. Сервер ответственен только за вызов этой функции и предполагает, что она существует на стороне клиента:

```
updateAuction({  
    "Title": "XBOX 360",  
    "Description": "Brand new XBOX 360 console",  
    "StartTime": "01/12/2012 12:00 AM",  
    "EndTime": "01/31/2012 12:00 AM",  
    "CurrentPrice": "$199.99",  
    "Bids": [  
        {  
            "Amount" : "$200.00",  
            "Timestamp": "01/12/2012 6:00 AM"  
        },  
        {  
            "Amount" : "$205.00",  
            "Timestamp": "01/14/2012 8:00 AM"  
        },  
        {  
            "Amount" : "$210.00",  
            "Timestamp": "01/15/2012 12:32 PM"  
        }  
    ]  
});
```

Очень важно отметить, что подход JSONP описывает другой метод обмена данными между клиентом и сервером. Вместо возвращения низкоуровневых данных JSON (как в случае нормального ответа AJAX) в ответе JSONP сервер помещает низкоуровневые данные JSON в оболочку вызова указанной функции клиентской стороны (в конце концов, ответы JSONP являются файлами сценариев JavaScript и соответственно они могут выполнять логику клиентской стороны). Таким образом, единственный способ доступа к данным, возвращаемым через вызов JSONP, является функция клиентской стороны — к ним нельзя обращаться напрямую.

В предыдущем примере ответ JSONP состоял только из обращения к функции обратного вызова, которой передавался сериализованный объект JSON. Тем не менее, ответ JSONP с тем же успехом мог бы выполнять другую логику перед функцией обратного вызова, например, осуществлять преобразование показаний времени с учетом часового пояса пользователя перед их отображением на экране:

```
var data = {
    "Title": "XBOX 360",
    "Description": "Brand new XBOX 360 console",
    "StartTime": "01/12/2012 12:00 AM",
    "EndTime": "01/31/2012 12:00 AM",
    "CurrentPrice": "$199.99",
    "Bids": [
        {
            "Amount" : "$200.00",
            "Timestamp": "01/12/2012 6:00 AM"
        },
        {
            "Amount" : "$205.00",
            "Timestamp": "01/14/2012 8:00 AM"
        },
        {
            "Amount" : "$210.00",
            "Timestamp": "01/15/2012 12:32 PM"
        }
    ]
};

/* Преобразовать показания времени в локальное время. */
function toLocalTime(src) {
    return new Date(src+" UTC").toString();
}

bid.StartTime = toLocalTime(bid.StartTime);
bid.EndTime = toLocalTime(bid.EndTime);

for(var i = 0; i < data.Bids.length; i++) {
    var bid = data.Bids[i];
    bid.Timestamp = toLocalTime(bid.Timestamp);
}

/* Выполнить обратный вызов. */
updateAuction(data);
```

Выполнение запроса JSONP

Метод `$.ajax` библиотеки jQuery предлагает первоклассную поддержку для запросов JSONP. Все, что понадобится сделать — это передать опции `dataType` и `jsonpCallback` для указания типа данных `jsonp` и имени функции обратного вызова клиентской стороны.

Выполнение запроса JSONP с помощью метода `$.ajax` библиотеки jQuery демонстрируется в следующем примере:

```
function updateAuction(result) {
    var message = result.Title + ": $" + result.CurrentPrice;
    $('#Result').html(message);
}
```

```
$.ajax({
    type: "GET",
    url: "http://localhost:11279/Auctions/Auction/1234",
    dataType: "jsonp",
    jsonpCallback: "updateAuction"
});
```

Обратите внимание, что поскольку это параметр строки запроса, а не действительная JavaScript-функция наподобие тех, что регистрируются для событий `.success()` и `.error()`, метод обратного вызова *должен* быть глобально доступной, уникально именованной функцией. В противном случае сценарий JSONP не сможет его выполнить.

Добавление поддержки JSONP к действиям контроллеров ASP.NET MVC

Инфраструктура ASP.NET MVC не имеет встроенной поддержки JSONP, поэтому для использования этого подхода потребуется самостоятельно реализовать все необходимое. К счастью, результаты JSONP – это не более чем модифицированная версия результата действия `JsonResult`, определенного в ASP.NET MVC Framework.

Пожалуй, наилучшим способом добавления поддержки JSONP к действиям контроллеров является создание специального класса `ActionResult`. Взгляните на пример 6.6.

Пример 6.6. JsonResult: специальный результат действия JSONP

```
using System.Web.Mvc;
public class JsonResult : JsonResult
{
    public string Callback { get; set; }
    public JsonResult()
    {
        JsonRequestBehavior = JsonRequestBehavior.AllowGet;
    }
    public override void ExecuteResult(ControllerContext context)
    {
        var httpContext = context.HttpContext;
        var callback = Callback;
        if(string.IsNullOrWhiteSpace(callback))
            callback = httpContext.Request["callback"];
        httpContext.Response.Write(callback + "(");
        base.ExecuteResult(context);
        httpContext.Response.Write(");");
    }
}
```

Вы могли заметить, что в классе `JsonResult` свойство `JsonRequestBehavior` жестко закодировано в `JsonRequestBehavior.AllowGet`. Причина в том, что по определению все запросы JSONP являются запросами GET.



Таким образом, каждый запрос JSONP подвержен упомянутой ранее уязвимости, поэтому следует избегать отправки конфиденциальной информации через JSONP!

Для предоставления ответа JSONP на запрос JSONP нужно просто возвратить экземпляр JsonpResult:

```
public ActionResult Auction(long id)
{
    var db = new DataContext();
    var auction = db.Auctions.Find(id);
    return new JsonpResult { Data = auction };
}
```

Включение разделения ресурсов между источниками

Предпочтительный метод для междоменных вызовов AJAX предусматривает использование *разделения ресурсов между источниками* (Cross-Origin Resource Sharing – CORS), если оно поддерживается. В отличие от JSONP, технология CORS не пользуется брешью в безопасности; вместо этого она использует специальный HTTP-заголовок, уведомляющий браузер о том, что сервер разрешает междоменные вызовы AJAX. Отсутствие “трюков” также делает подход CORS более простым, т.к. он не требует метода обратного вызова JavaScript или специального класса результата действия.

Чтобы включить поддержку CORS, установите значение заголовка Access-Control-Allow-Origin для каждого запроса, требующего CORS. В качестве значения этого заголовка можно указать “белый список” разрешенных доменов или просто “*” для выдачи разрешения на доступ из любого домена:

```
HttpContext.Response.AppendHeader("Access-Control-Allow-Origin", "*");
```

Включить CORS можно также и для всего приложения, добавив этот HTTP-заголовок в раздел конфигурации system.webServer⇒httpProtocol⇒customHeaders:

```
<system.webServer>
    <httpProtocol>
        <customHeaders>
            <add name="Access-Control-Allow-Origin" value="*" />
        </customHeaders>
    </httpProtocol>
</system.webServer>
```

После этого можно делать “нормальный” запрос с помощью метода \$.ajax() библиотеки jQuery:

```
$.ajax({
    type: "GET",
    url: "http://localhost:11279/Auctions/Auction/1234",
    dataType: "json",
    success: function (result) {
        var message = result.Title + ": $" + result.CurrentPrice;
        $('#Result').html(message);
    },
    error: function (XMLHttpRequest, textStatus, errorThrown) {
        alert("Error: " + errorThrown);
    }
});
```

Имея поддержку CORS, мы возвращаемся к простому и эффективному вызову AJAX, а все препятствия, которые приходилось преодолевать для выполнения запросов JSONP, остаются в прошлом.

Поддержка технологии CORS браузерами

На время написания данной книги технология Cross-Origin Resource Sharing (CORS) все еще находилась в черновой стадии, поэтому она не полностью поддерживается всеми основными браузерами. Следовательно, перед тем, как приступить к ее использованию, обязательно удостоверьтесь, что подход CORS работает с браузерами, на которые ориентирован ваш веб-сайт.

Резюме

Понимание того, когда и как использовать AJAX, является важным оружием разработчиков, которые хотят предложить своим пользователям более удобные интерфейсы. В этой главе были продемонстрированы различные способы применения AJAX для улучшения веб-приложений, а также представлен детальный обзор особенностей поддержки запросов AJAX в ASP.NET MVC. Кроме того, было показано, как с помощью мощных API-интерфейсов jQuery реализовать коммуникации AJAX на сайте.

ASP.NET Web API

С разрастанием пользовательского интерфейса приложения на стороне клиента за рамки нескольких простых запросов AJAX, может обнаружиться, что действия контроллеров, основанные на `JsonResult`, инфраструктуры ASP.NET MVC не полностью удовлетворяют нуждам расширенной интерфейсной части AJAX. Когда это происходит, может возникнуть необходимость в поиске более простого и элегантного пути для обработки расширенных запросов AJAX. Именно тогда можно приступить к использованию *ASP.NET Web API*.

Инфраструктура ASP.NET Web API Framework задействует как веб-стандарты, подобные HTTP, JSON и XML, так и стандартный набор соглашений для предоставления простого способа построения и открытия доступа к службам данных REST. С точки зрения архитектуры инфраструктура ASP.NET Web API очень похожа на ASP.NET MVC в том, что в ней применяются те же самые ключевые концепции, такие как маршрутизация, контроллеры и даже результаты действий контроллеров. Тем не менее, эти концепции используются для поддержки совершенно другого набора сценариев – сценариев, которые предусматривают работу с данными в противоположность генерации HTML-разметки.

В этой главе предлагается базовое введение в инфраструктуру ASP.NET Web API Framework, в котором будет показано, как создавать и открывать доступ к службам ASP.NET Web API, а затем потреблять эти службы через AJAX в браузере.

Построение службы данных

Добавление контроллера ASP.NET Web API в приложение в основном похоже на добавление контроллера ASP.NET MVC. В следующих разделах мы подробно рассмотрим этот процесс на примере добавления контроллера Web API в образцовое приложение Ebiz.

Перед тем как начать, понадобится папка, в которой будет храниться новый контроллер Web API. Контроллеры Web API могут находиться практически где угодно, так что только от вас зависит, каким соглашением при этом пользоватьсяся. Например, мы предпочитаем создавать новую папку по имени `Api` в корне веб-сайта, однако вы можете сохранять свои контроллеры Web API в папке `Controllers` прямо рядом с контроллерами ASP.NET MVC – при условии, что нет никаких конфликтов имен, инфраструктура ASP.NET способна легко их отличать.

Чтобы добавить новый контроллер Web API, просто щелкните правой кнопкой мыши на папке, которую необходимо добавить службу (в нашем случае это папка `Api`), и выберите в контекстном меню пункт `Add Controller...` (`Добавить Controller...`). Это приведет к открытию того же самого диалогового окна `Add Controller` (`Добавить`

контроллер), которое применялось для создания контроллеров ASP.NET MVC (рис. 7.1), но на этот раз в списке Template (Шаблон) потребуется выбрать вариант API controller with empty read/write actions (Контроллер Web API с пустыми действиями чтения/записи), а не один из шаблонов контроллеров ASP.NET MVC. (Дополнительные варианты этого списка были описаны в разделе “Шаблоны контроллера” главы 1.) Для начала новому контроллеру следует назначить имя. В этом примере будет использоваться имя AuctionsController.cs.

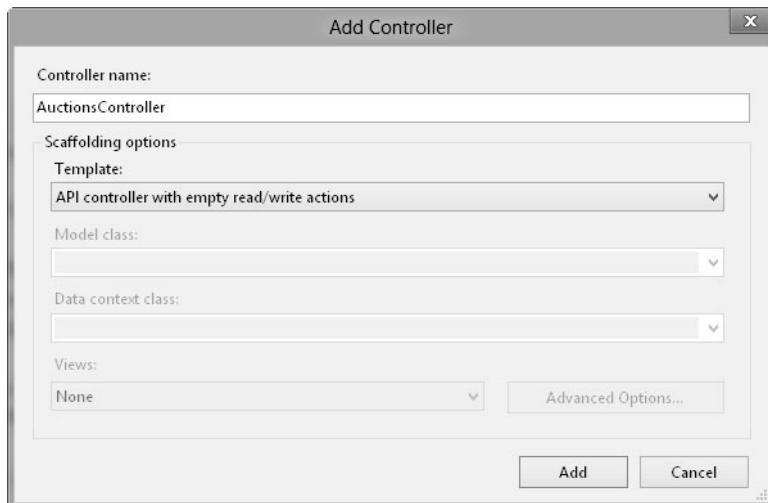


Рис. 7.1. Добавление контроллера Web API

По завершении щелкните на кнопке Add (Добавить) для добавления контроллера Web API в проект. Код нового контроллера Web API показан в примере 7.1.

Пример 7.1. Контроллер Web API

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Http;

namespace Ebuy.Website.Api
{
    public class AuctionsDataController : ApiController
    {
        // Запрос GET для api/auctions
        public IEnumerable<string> Get()
        {
            return new string[] { "value1", "value2" };
        }
        // Запрос GET для api/auctions/5
        public string Get(int id)
        {
            return "value";
        }
    }
}
```

```
// Запрос POST для api/auctions
public void Post(string value)
{
}

// Запрос PUT для api/auctions/5
public void Put(int id, string value)
{
}

// Запрос DELETE для api/auctions/5
public void Delete(int id)
{
}
}
```

Регистрация маршрутов Web API

Однако перед тем как этот новый контроллер можно будет использовать, он должен быть зарегистрирован с помощью инфраструктуры маршрутизации, чтобы он смог получать запросы.

Как и в случае ASP.NET MVC, запросы ASP.NET Web API основаны на URL с маршрутами к соответствующим действиям контроллеров. На самом деле маршруты ASP.NET Web API регистрируются *почти* так же, как маршруты ASP.NET MVC. Единственное отличие заключается в том, что вместо вспомогательного расширения `RouteTable.MapRoute()` в маршрутах Web API используется расширение `RouteTable.MapHttpRoute()`.

```
routes.MapHttpRoute(
    name: "DefaultApi",
    routeTemplate: "api/{controller}/{id}",
    defaults: new { id = RouteParameter.Optional }
);
```

Причина в том, что инфраструктура Web API Framework выясняет действие контроллера, которое должно быть выполнено, с применением *соглашения по конфигурации*.



Начинать свой маршрут не обязательно с литерального сегмента пути `api` – шаблон маршрутов Web API может быть изменен как угодно, при условии, что он не конфликтует с другими ресурсами, зарегистрированными в том же самом приложении.

Те же самые правила, которые применялись к маршрутизации ASP.MVC, также применяются и к службам данных – соблюдайте осторожность, чтобы маршруты не оказались слишком специфичными или чрезмерно неопределенными.

Соблюдение соглашения по конфигурации

Подобно ASP.NET MVC, в инфраструктуре ASP.NET Web API интенсивно используется соглашение по конфигурации, что позволяет облегчить рабочую нагрузку, связанную с созданием служб данных. Например, вместо обязательного аннотирования каждого метода с помощью такого атрибута, как `HttpPostAttribute`, для идентификации типа запросов, обрабатываемых действием (что должно делаться в случае действий контроллеров ASP.NET MVC), методы `ApiController` полагаются на имена, которые соответствуют стандартным действиям HTTP.

Это соглашение значительно упрощает выполнение операций CRUD (Create, Read, Update, Delete – создание, чтение, обновление, удаление) над ресурсом (сущностью). Ниже перечислены стандартные действия HTTP и соответствующие им операции CRUD.

- **GET (чтение).** Извлекает представление ресурса.
- **PUT (обновление).** Обновляет существующий ресурс (или создает новый экземпляр).
- **POST (создание).** Создает новый экземпляр ресурса.
- **DELETE (удаление).** Удаляет ресурс.



Метод PUT будет заменять полную сущность. Для поддержки частичного обновления вместо него необходимо применять метод PATCH.

Взаимодействовать со службой данных ASP.NET Web API невероятно просто. Например, в приведенном далее фрагменте демонстрируется использование метода `$.getJSON()` библиотеки jQuery для выполнения запроса GET к службе `/api/auction`, которая возвращает коллекцию аукционных товаров, сериализованных в формате JSON:

```
<script type="text/javascript">
$(function () {
    $.getJSON("api/auction/",
        function (data) {
            $.each(data, function (key, val) {
                var str = val.Description;
                $('<li/>', { html: str }).appendTo($('#auctions'));
            });
        });
    });
</script>
```

Переопределение соглашений

Важно отметить, что соглашение по именованию действий контроллеров применяется, только когда имя соответствует одному из стандартных действий REST (GET, POST, PUT и DELETE). Тем не менее, если вы хотите именовать свои методы по-другому, но по-прежнему пользоваться остальной функциональностью Web API, можете применить атрибут `AcceptVerbsAttribute` – либо его псевдонимы, такие как `HttpGetAttribute` или `HttpPostAttribute` – к методам контроллера Web API, в точности как применялся бы атрибут к действию контроллера ASP.NET MVC.

Сказанное демонстрируется в следующем фрагменте кода:

```
[HttpGet]
public Auction FindAuction(int id)
{
}
```

В этом примере мы решили нарушить соглашение REST и назвать действие контроллера `FindAuction` вместо использования имени метода `Get`, как того требует соглашение. В таком случае мы применили к действию контроллера `FindAuction` атрибут `HttpGetAttribute`, указав, что это действие обрабатывает запросы GET.

Подключение Web API

А теперь рассмотрим установку созданного ранее контроллера Web API, чтобы он мог выполнять операции CRUD над аукционными товарами.

Для доступа в базу данных Entity экземпляр класса репозитория данных передается конструктору AuctionsDataController:

```
public class AuctionsDataController : ApiController
{
    private readonly IRepository _repository;
    public AuctionsDataController(IRepository repository)
    {
        _repository = repository;
    }
}
```

По умолчанию контроллеры Web API требуют наличия стандартного (без параметров) конструктора. Поскольку контроллеру должен передаваться IRepository, потребуется инициализировать специальный класс распознавателя зависимостей во время начальной загрузки приложения:

```
GlobalConfiguration.Configuration.DependencyResolver =
    new NinjectWebApiResolver(kernel);
```

Ниже представлен пример специального распознавателя зависимостей, который использует контейнер Ninject IoC. Учитывая, что контроллеры Web API создаются для каждого запроса, специальный распознаватель должен создавать новую область зависимости (например, NinjectWebApiScope) для каждого запроса:

```
using System.Web.Http.Dependencies;
using Ninject;

public class NinjectWebApiResolver : NinjectWebApiScope,
    IDependencyResolver
{
    private IKernel kernel;
    public NinjectWebApiResolver(IKernel kernel) : base(kernel)
    {
        this.kernel = kernel;
    }
    public IDependencyScope BeginScope()
    {
        return new NinjectWebApiScope(kernel.BeginBlock());
    }
}
```

Код специального класса области зависимости Ninject приведен ниже. Когда запрашивается контроллер Web API, будет вызван метод GetService(); метод Resolve() поддерживает внедрение репозитория при создании экземпляра контроллера:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web.Http.Dependencies;
using Ninject.Activation;
using Ninject.Parameters;
using Ninject.Syntax;
```

```

public class NinjectWebApiScope : IDependencyScope
{
    protected IResolutionRoot resolutionRoot;
    public NinjectWebApiScope(IResolutionRoot resolutionRoot)
    {
        this.resolutionRoot = resolutionRoot;
    }
    public object GetService(Type serviceType)
    {
        return resolutionRoot.Resolve(
            this.CreateRequest(serviceType)).SingleOrDefault();
    }
    public IEnumerable<object> GetServices(Type serviceType)
    {
        return resolutionRoot.Resolve(this.CreateRequest(serviceType));
    }
    private IRequest CreateRequest(Type serviceType)
    {
        return resolutionRoot.CreateRequest(serviceType,
            null,
            new Parameter[0],
            true,
            true);
    }
    public void Dispose()
    {
        resolutionRoot = null;
    }
}

```

В следующем коде показан полностью реализованный контроллер Web API, который был модифицирован с целью работы с репозиторием для выполнения операций CRUD на классе Auctions:

```

public class AuctionsDataController : ApiController
{
    private readonly IRepository _repository;
    public AuctionsDataController(IRepository repository)
    {
        _repository = repository;
    }
    public IEnumerable<Auction> Get()
    {
        return this._repository.All<Auction>();
    }
    public Auction Get(string id)
    {
        return _repository.Single<Auction>(id);
    }
    public void Post(Auction auction)
    {
        _repository.Add<Auction>(auction);
    }
}

```

```

public void Put(string id, Auction auction)
{
    var currentAuction = _repository.Single<Auction>(id);
    if (currentAuction != null)
    {
        currentAuction = Mapper.DynamicMap<Auction>(auction);
    }
}
public void Delete(string id)
{
    _repository.Delete<Auction>(id);
}
}

```

Разбиение на страницы и запрашивание данных

Одним из наиболее мощных аспектов инфраструктуры ASP.NET Web API Framework является поддержка разбиения на страницы и фильтрации данных через веб-протокол Open Data Protocol (OData) для запросов с выражениями в параметрах URL. Например, URI вида /api/Auction?\$top=3&\$orderby=CurrentBid возвращает три верхних аукционных товара, упорядоченные по значению их свойства CurrentBid.

В табл. 7.1 перечислены распространенные параметры запросов, воспринимающие OData.

Таблица 7.1. Параметры строки запроса, поддерживающие OData

Параметр строки запроса	Описание	Пример
\$filter	Фильтрует сущности, которые соответствуют булевскому выражению	/api/Auction?\$filter=CurrentBid gt 2
\$orderby	Возвращает группу сущностей, упорядоченных по заданному полю	/api/Auction?\$orderby=Description
\$skip	Пропускает первые <i>n</i> сущностей	/api/Auction?\$skip=2
\$top	Возвращает первые <i>n</i> сущностей	/api/Auction?\$top=3



Подробные сведения о спецификации Open Data Protocol (OData) доступны на веб-сайте OData по адресу <http://www.odata.org/>.

Для поддержки разбиения на страницы и фильтрации действие контроллера Web API должно возвращать результат `IQueryable<T>`. Если данные не являются `IQueryable<T>`, можно воспользоваться LINQ-расширением `AsQueryable()`. Затем Web API берет результат `IQueryable<T>` и преобразует строку запроса OData в выражение LINQ, которое будет применяться для фильтрации элементов в коллекции `IQueryable<T>`.

Далее инфраструктура Web API Framework преобразует выражение LINQ в объект JSON, который доставляется посредством результата HTTP:

```
public IQueryable<Auction> Get()
{
    return _repository.All<Auction>().AsQueryable();
}
```

Обработка исключений

Разработчики, строящие приложения на основе AJAX, должны проявлять особую осторожность при обработке исключений. По умолчанию если на сервере во время обработки запроса AJAX возникает ошибка, возвращается внутренняя ошибка сервера (500). Это приводит к возникновению ряда проблем.

Прежде всего, сообщение пользователю о том, что возникла внутренняя ошибка сервера, не является особенно полезным. Кроме того, возвращение сообщений об ошибках предлагает разработчикам только ограниченную информацию для проведения отладки и отслеживания проблемы. Однако важнее всего то, что отправка пользователям сообщений без определенной предварительной "очистки" потенциально представляет собой серьезный риск в плане безопасности: сообщение об ошибке может содержать стек вызовов или другие сведения, которыми злоумышленник может воспользоваться для компрометации вашего сайта!

На рис. 7.2 показан пример внутренней ошибки сервера, возвращенной из контроллера Web API. Сообщение об ошибке не содержит ничего полезного кроме стека вызовов.

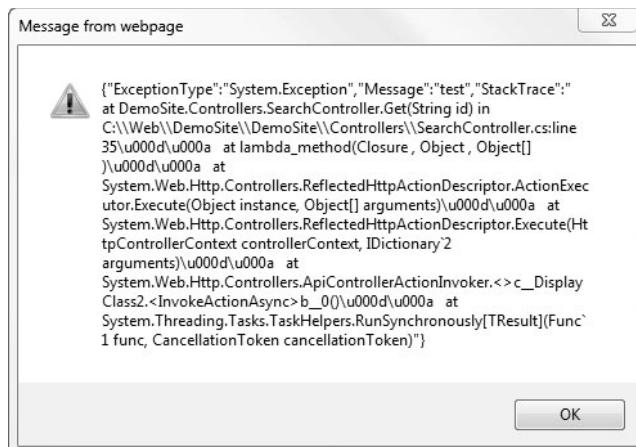


Рис. 7.2. Внутренняя ошибка сервера

К счастью, ASP.NET Web API предлагает ряд возможностей для обработки исключений и возврата более значащей информации клиентскому приложению. Например, класс `HttpResponseException` дает намного больший контроль над кодами состояния HTTP и сообщениями ответов, отправляемыми клиенту, чем традиционные подходы к обработке ошибок.

В следующем примере демонстрируется применение класса `HttpResponseException` для установки кода состояния HTTP 404 и настройки сообщения об ошибке, возвращаемого клиенту:

```
public Auction Get(string id)
{
    var result = _repository.Single<Auction>(id);
    if (result == null)
    {
        var errorMessage = new HttpResponseMessage(HttpStatusCode.NotFound);
        errorMessage.Content = new StringContent
            (string.Format("Invalid id, no auction available for id: {0}.", id));
        errorMessage.ReasonPhrase = "Not Found";
        throw new HttpResponseException(errorMessage);
    }
    return result;
}
```

В дополнение к использованию `HttpResponseException`, инфраструктура ASP.NET WEB API позволяет создавать фильтры исключений. Фильтры исключений вызываются, когда в контроллере были сгенерированы необработанные исключения, не относящиеся к типу `HttpResponseException`.

Чтобы создать фильтр исключений, можно напрямую реализовать интерфейс `System.Web.Http.Filters.IExceptionFilter` или унаследовать его от `ExceptionFilterAttribute`. Создание специального атрибута является простым и предпочтительным способом создания фильтров исключений. Этот подход требует только переопределения метода `OnException()`:

```
using System.Diagnostics;
using System.Web.Http.Filters;
public class CustomExceptionFilter : ExceptionFilterAttribute
{
    public override void OnException(HttpActionExecutedContext context)
    {
        base.OnException(context);
    }
}
```

Допускается также переопределение ответа HTTP, отправляемого обратно клиенту. Это можно сделать путем модификации параметра `HttpActionExecutedContext`:

```
using System.Web.Http.Filters;
using System.Net.Http;
using System.Net;
public class CustomExceptionFilter : ExceptionFilterAttribute
{
    public override void OnException(HttpActionExecutedContext context)
    {
        if (context.Response == null)
        {
            context.Response = new HttpResponseMessage();
        }
        context.Response.StatusCode = HttpStatusCode.NotImplemented;
        context.Response.Content = new StringContent("Custom Message");
        base.OnException(context);
    }
}
```

Сразу после создания специальный фильтр исключений должен быть зарегистрирован. Существуют два способа регистрации фильтра исключений: его можно зарегистрировать глобально, добавив в коллекцию `GlobalConfiguration.Configuration.Filters`, или же добавить его как атрибут к методу контроллера Web API. Глобально зарегистрированные фильтры будут выполняться для всех генерируемых исключений кроме `HttpResponseException`, во всех контроллерах Web API.

Регистрация глобальных фильтров исключений осуществляется просто – нужно лишь добавить любые специальные фильтры в коллекцию `GlobalConfiguration.Configuration.Filters` во время фазы начальной загрузки приложения:

```
public class MvcApplication : System.Web.HttpApplication
{
    static void ConfigureApi(HttpConfiguration config)
    {
        config.Filters.Add(new CustomExceptionFilter());
    }

    protected void Application_Start()
    {
        ConfigureApi(GlobalConfiguration.Configuration);
    }
}
```

В качестве альтернативы метод контроллера Web API может быть аннотирован напрямую с применением атрибута специального фильтра исключений:

```
[CustomExceptionFilter]
public Auction Get(string id)
{
    var result = _repository.Single<Auction>(id);
    if (result == null)
    {
        throw new Exception("Item not Found!");
    }
    return result;
}
```

Фильтры исключений ASP.NET Web API похожи на фильтры ASP.NET MVC с той лишь разницей, что они определены в другом пространстве имен и ведут себя слегка иначе. Например, класс `HandleErrorAttribute` инфраструктуры ASP.NET MVC не может обрабатывать исключения, сгенерированные контроллерами Web API.

Форматеры носителей

Одним из более мощных аспектов инфраструктуры ASP.NET Web API является возможность работать со многими разными типами носителей (типами MIME). Типы MIME используются для описания формата данных в запросе HTTP. Тип MIME состоит из двух строк, типа и подтипа: к примеру, `text.html` применяется для описания формата HTML.

Клиент может установить HTTP-заголовок `Accept` для сообщения серверу, какие типы MIME он желает получать обратно. Скажем, следующий заголовок `Accept` указывает серверу, что клиент ожидает возврата либо HTML, либо XHTML:

```
Accept: text/html,application/xhtml+xml,application
```

Инфраструктура ASP.NET Web API применяет тип носителя для определения способа сериализации и десериализации тела сообщения HTTP. Имеется готовая поддержка для XML, JSON и данных, закодированных в форме.

Чтобы получить специальный форматер носителя, просто создайте класс, производный от `MediaTypeFormatter` или `BufferedMediaTypeFormatter`. Класс `MediaTypeFormatter` использует подход с асинхронным чтением и записью; класс `BufferedMediaTypeFormatter` унаследован от `MediaTypeFormatter` и предлагает оболочки в виде синхронных операций для методов асинхронного чтения и записи. Хотя наследовать от класса `BufferedMediaTypeFormatter` проще, в этом случае может возникнуть проблема блокирования потоков.

В следующем примере показано, как создать специальный тип носителя для сериализации элемента `Auction` в формат разделяемых запятыми значений (CSV). Для простоты специальный форматер унаследован от `BufferedMediaTypeFormatter`. В конструкторе форматера должны быть определены поддерживаемые типы носителей:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Net;
using System.Net.Http.Formatting;
using System.Net.Http.Headers;
using DemoSite.Models;

public class AuctionCsvFormatter : BufferedMediaTypeFormatter
{
    public AuctionCsvFormatter()
    {
        thisSupportedMediaTypes.Add(new MediaTypeHeaderValue("text/csv"));
    }
}
```

Для сериализации и десериализации сущностей понадобится переопределить методы `CanWriteType()` и `CanReadType()`. Эти методы применяются для определения типов, которые поддерживаются специальным форматером:

```
protected override bool CanWriteType(Type type)
{
    if (type == typeof(Auction))
    {
        return true;
    }
    else
    {
        Type enumerableType = typeof(IEnumerable<Auction>);
        return enumerableType.IsAssignableFrom(type);
    }
}

protected override bool CanReadType(Type type)
{
    return false;
}
```

Когда форматер выполняется, метод `OnWriteToStream()` вызывается для сериализации типа в поток, а метод `OnReadFromStream()` – для десериализации типа из потока.

В примере 7.2 показано, как сериализовать одиночный тип Auction или коллекцию типов Auction. Обратите внимание, что метод Encode отменяет некоторые символы. Это важный момент, который следует иметь в виду при работе со специальными форматерами.

Пример 7.2. Сериализация типа

```
protected override void OnWriteToStream(Type type,
    object value, Stream stream,
    HttpContentHeaders contentHeaders,
    FormatterContext formatterContext,
    TransportContext transportContext)
{
    var source = value as IEnumerable<Auction>;
    if (source != null)
    {
        foreach (var item in source)
        {
            WriteItem(item, stream);
        }
    }
    else
    {
        var item = value as Auction;
        if (item != null)
        {
            WriteItem(item, stream);
        }
    }
}

private void WriteItem(Auction item, Stream stream)
{
    var writer = new StreamWriter(stream);
    writer.WriteLine("{0},{1},{2}",
        Encode(item.Title),
        Encode(item.Description),
        Encode(item.CurrentPrice.Value));
    writer.Flush();
}

static char[] _specialChars = new char[] { ',', '\n', '\r', '"' };

private string Encode(object o)
{
    string result = "";
    if (o != null)
    {
        string data = o.ToString();
        if (data.IndexOfAny(_specialChars) != -1)
        {
            result = String.Format("\"{0}\"", data.Replace("\"", "\"\""));
        }
    }
    return result;
}
```

Перед использованием специальный форматер носителя должен быть зарегистрирован. Добавьте созданный специальный форматер носителя в коллекцию GlobalConfiguration.Configuration.Filters внутри метода Application_Start() в Global.asax.cs:

```
static void ConfigureApi(HttpConfiguration config)
{
    config.Formatters.Add(new AuctionCsvFormatter());
}
```

После регистрации специальный форматер носителя будет выполняться для любого запроса, содержащего заголовок text/csv Accept.

Резюме

В этой главе было дано введение в новую инфраструктуру ASP.NET Web API Framework, предлагаемую Microsoft, и показано, как ее использовать в качестве простого способа открытия доступа к службам данных для веб-приложений.

Расширенная работа с данными

До сих пор основное внимание в книге было сконцентрировано на ключевых компонентах ASP.NET MVC: модели, представлении и контроллере. В этой главе мы переключимся на уровень доступа к данным и покажем, как использовать репозиторий и объектно-реляционное отображение при построении веб-приложений ASP.NET MVC.

Шаблоны доступа к данным

Одной из ключевых характеристик инфраструктуры ASP.NET MVC Framework является расширяемость. Эта инфраструктура была спроектирована для предоставления разработчикам высокой гибкости в плане подключения различных компонентов. С учетом того, что ASP.NET MVC построена поверх .NET 4.5, при разработке уровня доступа к данным в приложении могут использоваться любые популярные инфраструктуры доступа к данным, включая ADO.NET, LINQ to SQL, ADO.NET Entity Framework и NHibernate.

Независимо от выбранной инфраструктуры доступа к данным, важно понимать ключевые шаблоны проектирования для доступа к данным, которые дополняют шаблон “модель-представление-контроллер”.

Традиционные объекты CLR

Традиционный объект CLR (Plain Old CLR Object – РОСО) – это класс .NET, который используется для представления класса бизнес-сущности (модели). Этот класс фокусируется на ключевых бизнес-атрибутах (свойства) и поведении (методы) бизнес-сущности и связанных с ней сущностей, не требуя никакого специфичного для базы данных кода.

Основная цель использования классов РОСО заключается в проектировании бизнес-модели приложения с *игнорированием постоянства* (persistence ignorance – РИ). Такой подход к проектированию позволяет бизнес-модели приложения развиваться независимо от модели доступа к данным. Поскольку бизнес-модель не содержит кода доступа к данным, облегчается тестирование модели в изоляции, и лежащие в основе данные могут легко быть заменены в случае изменения бизнес-требований.

Ниже приведен пример класса РОСО, содержащего только свойства и методы:

```
public class Product
{
    public long Id { get; set; }
```

```

public string Name { get; set; }
public double Price { get; set; }
public int NumberInStock { get; set; }
public double CalculateValue()
{
    return Price * NumberInStock;
}
}

```

Обратите внимание, что этот класс не содержит кода, специфичного для инфраструктуры базы данных. Позже в этой главе мы покажем, как использовать объектно-реляционный отображатель (ORM) и шаблон Repository (Репозиторий) для обеспечения постоянства классов POCO.

Бизнес-модель для приложения вполне может содержать десятки или даже сотни классов в более сложных сценариях. Даже в случае простых моделей, которые включают лишь несколько классов, имеет смысл создать базовый сущностный класс, содержащий общие свойства и методы.

Ниже показан пример базового сущностного класса. Обратите внимание, что класс и его методы помечены как `abstract`; это общий подход к обеспечению согласованности по всей модели в базовом сущностном классе:

```

public abstract class BaseEntity
{
    public string Key { get; set; }
    public abstract void GenerateKey();
}

```

Использование шаблона Repository

Шаблон *Repository* (*Репозиторий*) – это шаблон доступа к данным, который способствует слабо связанному подходу в отношении доступа к данным. Вместо помещения логики доступа к данным в контроллер или бизнес-модель, ответственность за постоянство бизнес-модели приложения возлагается на отдельный класс или набор классов, называемых *репозиторием*.

Шаблон *Repository* хорошо дополняет ключевой принцип проектирования шаблона MVC – разделение ответственности. За счет применения этого шаблона мы изолируем уровень доступа к данным от остальной части приложения и получаем все преимущества от использования классов POCO.

Для проектирования репозитория существуют разные подходы, которые кратко описаны ниже.

- Один на бизнес-модель.** Наиболее прямолинейный способ заключается в создании репозитория для каждого класса бизнес-модели. Хотя этот подход и прост, он может приводить к проблемам, таким как дублирование кода или сложность, когда множеству репозиториев понадобится взаимодействовать друг с другом.
- Использование агрегированного корня.** Агрегированный корень – это класс, который может существовать сам по себе и является ответственным за управление ассоциациями с другими связанными классами. Например, в приложении электронной коммерции может существовать класс `OrderRepository`, который будет обрабатывать создание заказа и связанных с ним элементов в заказе.
- Обобщенный репозиторий.** Вместо создания специфических классов репозитория разработчик может воспользоваться обобщениями .NET для построения общего репозитория, который будет применяться множеством приложений. Образцовое приложение `Ebu` включает пример *обобщенного репозитория*.

Ниже представлена базовая структура класса репозитория:

```
public class ModelRepository
{
    public ModelRepository()
    {
    }

    public void Add(Model instance)
    {
    }

    public void Update(Model instance)
    {
    }

    public void Delete(Model instance)
    {
    }

    public Model Get(string id)
    {
    }

    public ICollection<Model> GetAll()
    {
    }
}
```

В дополнение к выполнению операций CRUD (Create, Read, Update, Delete – создание, чтение, обновление, удаление) для сущности, репозиторий иногда отвечает за кеширование сущностей. Кеширование великолепно работает для сущностей, которые в основном являются статическими, такие как поисковые значения для раскрывающихся списков, но может оказаться проблематичным для часто обновляемых сущностей.

Дополнительные сведения о кешировании данных представлены в главе 12.

В рамках ASP.NET MVC контроллеры взаимодействуют с репозиториями для загрузки и сохранения бизнес-модели приложения. За счет использования технологии внедрения зависимостей (DI) репозитории могут внедряться через конструктор контроллера. На рис. 8.1 изображено отношение между репозиторием и контекстом данных Entity Framework, при котором контроллеры ASP.NET MVC взаимодействуют с репозиторием, а не напрямую с Entity Framework.

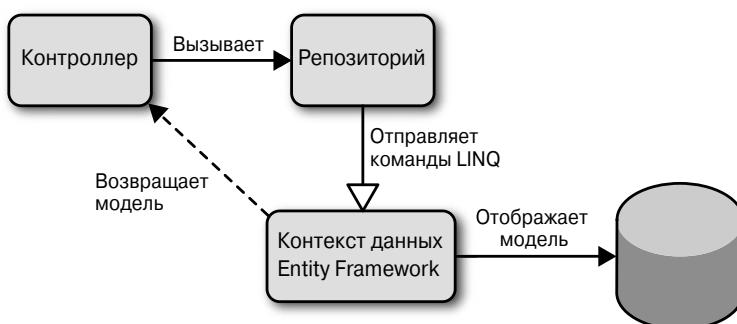


Рис. 8.1. Организация взаимодействий при наличии репозитория

В следующем примере показано, как репозиторий внедряется в контроллер с применением технологии внедрения зависимостей, и демонстрируется использование внедренного репозитория для извлечения списка аукционных товаров. Применение внедрения зависимостей упрощает тестирование контроллера за счет имитации передаваемого репозитория:

```
public class AuctionsController : Controller
{
    private readonly IRepository _repository;
    public AuctionsController(IRepository repository)
    {
        _repository = repository;
    }
    public ActionResult Index()
    {
        var auctions = _repository.GetAll<Auction>();
        return auctions;
    }
}
```

Объектно-реляционные отображатели

Объектно-реляционный отображатель (object relational mapper – ORM) – это шаблон доступа к данным, который поддерживает отображение сущностей между классами (типами .NET Framework) и моделью реляционной базы данных. Главная причина использования этого шаблона объясняется наличием значительного разрыва между структурой бизнес-модели приложения и моделью базы данных. Такой разрыв называется *объектно-реляционной потерей соответствия* (object relational impedance mismatch); это является просто причудливым способом сказать о том, что лучшие структуры для бизнес-уровня и уровня доступа к данным в приложении обычно не совместимы.



Просто отражая модель реляционной базы данных в бизнес-уровне, легко угодить в ловушку. Проблема такого подхода состоит в том, что он ограничивает возможности по использованию всей мощности платформы .NET.

Ниже перечислены основные проблемные точки объектно-реляционной потери соответствия.

- **Степень детализации.** Иногда модель будет содержать больше классов, чем количество соответствующих таблиц в базе данных. Хорошим примером является класс `Address`, поскольку с различными видами адресов, существующих в реальном мире, может быть связано разное поведение – к примеру, платежный адрес может обрабатываться иначе, чем адрес доставки. Хотя зачастую удобно представлять эти отличия с применением разных классов, каждый со своим специфическим набором поведений, все они могут содержать (в основном) одни и те же данные, поэтому может иметь смысл хранить все типы `Address` в единственной таблице базы данных.
- **Наследование.** Концепция наследования – или создания классов, производных от других классов с целью разделения общей логики – представляет собой один из наиболее важных аспектов объектно-ориентированной разработки. Однако, невзирая на важность объектно-ориентированной разработки, реляционные базы данных, как правило, не воспринимают концепцию наследования.

Например, в то время как реляционная база данных имеет единственную таблицу *Customers* со специальным столбцом, который определяет, является заказчик внутренним или международным, модель предметной области может выражать это отношение через базовый класс *Customer* и несколько подклассов, таких как *DomesticCustomer* и *InternationalCustomer*, для представления различных видов заказчиков, с которыми приходится иметь дело.

- **Идентичность.** Реляционные базы данных полагаются на наличие одиночного столбца (т.е. *первичного ключа* таблицы), выступающего в качестве уникального идентификатора для каждой строки. Это часто конфликтует с миром .NET Framework, в котором объекты могут распознаваться как по идентичности ($a == b$), так и по равенству ($a.Equals(b)$), причем в обоих случаях не предполагается, что объект должен иметь одиночное уникальное свойство или поле.
- **Ассоциации.** Реляционная база данных использует первичный и внешний ключи для установки ассоциаций между сущностями. В то же время платформа .NET Framework представляет объектные ассоциации как односторонние ссылки. Например, в реляционной базе данных допускается запрашивать данные из таблиц в любом направлении. Однако в .NET ассоциацией владеет только один класс, поэтому для получения двунаправленной поддержки потребуется копировать ассоциацию. Кроме того, невозможно узнать множественность ассоциации в классе. Концепции “один ко многим” и “многие ко многим” являются неотличимыми.
- **Навигация по данным.** Способ доступа к данным в классе .NET Framework фундаментально отличается от способа, применяемого в реляционной базе данных. В модели предметной области, реализованной с использованием .NET, вы переходите с одного связанного отношения к другому по всему объектному графу модели, в то время как обычно пытаетесь минимизировать количество SQL-запросов, требуемых при загрузке множества сущностей с помощью конструкций *JOIN* и специфических операторов *SELECT*.

Хотя для выполнения таких операций доступа к данным, как загрузка и сохранение данных, разработчики могут применять модель, важность и ответственность базы данных по-прежнему имеют первостепенное значение. Традиционные правила проектирования доступа к данным должны быть обязательно соблюдены. Каждая таблица должна иметь единственный первичный ключ, отношения “один ко многим” должны определяться с использованием внешних ключей и т.д. Например, для представления отношения между студентами и преподавателями понадобится третья таблица (скажем, *Class* (занятие)), т.к. студенты и преподаватели могут иметь одно или более занятий.

Обзор Entity Framework

Создание ORM вручную может оказаться непосильной задачей. К счастью, вместо самостоятельного построения всего необходимого разработчики могут обратиться к одной из многочисленных доступных инфраструктур ORM. В их число входит пара, предлагаемая Microsoft – LINQ to SQL и ADO.NET Entity Framework – и много популярных коммерческих и с открытым кодом инфраструктур от других поставщиков, таких как NHibernate (<http://www.nhibernate.com>).

Инфраструктура ADO.NET Entity Framework (EF) – это объектно-реляционный отображатель, входящий в состав платформы .NET. При использовании EF разработчик

взаимодействует с существенной моделью, а не с моделью реляционной базы данных приложения. Эта абстракция позволяет разработчику сосредоточиться на бизнес-поведении и отношениях между сущностями, вместо того, чтобы заниматься деталями сохранения сущностей в модели реляционных данных. Для взаимодействия с существенной моделью разработчик использует контекст данных Entity Framework, чтобы выдавать запросы или сохранять модель. При вызове одной из таких операций EF будет генерировать необходимый код SQL для выполнения операции.

Одним из наиболее спорных вопросов при переходе от традиционных подходов доступа к данным (к примеру, объекты *Dataset* из ADO.NET) на подход объектно-реляционного отображения является определение роли, которую должны играть хранимые процедуры, если они предусмотрены. Поскольку во время применения ORM основное внимание уделяется существенной модели, разработчикам рекомендуется позволить инфраструктуре обрабатывать отображение сущностей, а не беспокоиться написанием SQL-запросов самостоятельно.

К счастью, если вы работаете в организации, которая требует написания хранимых процедур, или необходимо взаимодействовать с существующей базой данных, в которой используются хранимые процедуры, то инфраструктура ADO.NET Entity Framework предлагает поддержку вызова таких процедур. За дополнительными сведениями обращайтесь по адресу:

[http://msdn.microsoft.com/ru-ru/library/bb399203\(v=vs.90\).aspx](http://msdn.microsoft.com/ru-ru/library/bb399203(v=vs.90).aspx)

Ниже приведен пример применения контекста данных Entity Framework для добавления и сохранения новых товаров. Когда метод *SaveChanges()* вызывается, Entity Framework генерирует код SQL для вставки двух новых товаров:

```
using (var db = new ProductModelContainer())
{
    db.Product.Add(new Product { Id = "173", Name = "XBox 360" });
    db.Product.Add(new Product { Id = "225", Name = "Halo 4" });
    db.SaveChanges()
}
```

Выбор подхода доступа к данным

В Microsoft признают, что единый подход для доступа к данным не работает. Некоторые разработчики отдают предпочтение подходу, более ориентированному на данные, который сосредоточен на проектировании базы данных первой с последующей генерацией для приложения бизнес-модели, во многом управляемой структурой базы данных. Другие разработчики желают использовать классы РОСО для определения структуры бизнес-модели и либо генерации базы данных из классов РОСО, либо привязки модели к существующей базе данных.

В результате ADO.NET Entity Framework позволяет разработчикам осуществлять выбор между тремя разными стилями.

- **Database First (Сначала база данных).** Для разработчиков, предпочитающих применять более ориентированный на данные подход или начинать с существующей базы данных, Entity Framework предлагает возможность генерировать бизнес-модель на основе таблиц и их строк в реляционной базе данных. Инфраструктура Entity Framework использует специальный конфигурационный файл (с расширением .edmx) для хранения информации о схеме базы данных, концептуальной модели данных и сведений об отображении между ними.

Разработчики могут применять визуальный конструктор Entity Framework, включенный в Visual Studio, для генерации, отображения и редактирования концептуальной модели, используемой Entity Framework.

- **Model First (Сначала модель).** Для разработчиков, не имеющих существующей базы данных, Entity Framework предлагает визуальный конструктор, который может применяться для создания концептуальной модели данных. Как и в случае подхода Database First, инфраструктура Entity Framework использует файл схемы для хранения информации, связанной с отображением модели на схему базы данных. После того как модель создана, визуальный конструктор EF может сгенерировать схему базы данных, посредством которой можно создать саму базу данных.
- **Code First (Сначала код).** Разработчики, желающие применять подход с большим игнорированием постоянства, могут создавать бизнес-модель прямо в коде. Инфраструктура Entity Framework предоставляет специальный API-интерфейс отображения и поддерживает набор соглашений, обеспечивающих работу этого подхода. При подходе Code First инфраструктура Entity Framework не использует внешний конфигурационный файл для хранения схемы базы данных, поскольку API-интерфейс отображения применяет эти соглашения для генерации схемы базы данных динамически во время выполнения.



В настоящее время подход Entity Framework Code First не поддерживает отображение на хранимые процедуры. Для выполнения хранимых процедур могут использоваться методы `ExecuteSqlCommand()` и `SqlQuery()`.

Параллелизм базы данных

Обработка *конфликтов параллелизма* является одним из наиболее важных аспектов, который разработчики должны учитывать при построении веб-приложений. Конфликты параллелизма возникают, когда множество пользователей одновременно пытаются изменить одни и те же данные. По умолчанию, если только инфраструктура Entity Framework не сконфигурирована на обнаружение конфликтов, применяется правило “выигрывает последний”. Например, если два пользователя одновременно загружают один и тот же товар для редактирования, то выиграет тот из них, который щелкнет на кнопке отправки последним, и данные этого пользователя перезапишут данные другого пользователя без выдачи какого-либо предупреждения любому из них.

В зависимости от типа приложения и изменчивости его данных, разработчик принимает решение относительно того, перевешивают ли затраты на программирование с учетом параллелизма получаемые в результате преимущества. Для поддержки параллелизма существуют два подхода.

- **Пессимистический параллелизм.** Подход пессимистического параллелизма требует использования блокировок базы данных, чтобы предотвратить перезаписывание множеством пользователей изменений друг у друга. Когда извлекается строка данных, применяется блокировка только для чтения, которая удерживается до тех пор, пока этот же пользователь не обновит данные или не удалит блокировку только для чтения. Такой подход может вызвать множество проблем в веб-приложениях, т.к. веб-сеть в значительной степени полагается на модель без поддержки состояния. Основная проблема, на которую следует обратить внимание, касается управления после того, как блокировки только для чтения

удалены; поскольку для доступа к веб-приложению применяется веб-браузер, нет никаких гарантий того, что пользователь когда-либо выполнит действие, которое может запустить удаление блокировки только для чтения.

- **Оптимистический параллелизм.** Вместо того чтобы полагаться на блокировки базы данных, подход оптимистического параллелизма предусматривает проверку, что обновляемые данные не были модифицированы с момента их первоначального извлечения. Это обычно достигается добавлением в таблицу поля метки времени, в котором отслеживается время последнего обновления строки. Затем перед применением любых обновлений к строке приложение проверяет это поле на предмет того, изменялась ли строка кем-нибудь с момента извлечения пользователем данных.

Инфраструктура ADO.NET Entity Framework не предлагает готовой поддержки для пессимистического параллелизма. Вместо этого рекомендуется использовать оптимистический параллелизм. В Entity Framework поддерживаются два способа применения оптимистического параллелизма: добавление свойства метки времени к сущности и обработка любых исключений `OptimisticConcurrencyException`, возвращаемых из контекста данных Entity Framework.

Ниже представлен пример использования атрибута `Timestamp` для добавления свойства метки времени к сущности. В результате применения этого атрибута соответствующий столбец базы данных будет добавлен в виде условия к SQL-конструкции `Where` при выполнении любой операции `UPDATE` или `DELETE`:

```
[Timestamp]  
public Byte[] Timestamp { get; set; }
```

Для перехвата ошибок `OptimisticConcurrencyException` применяйте обычные приемы .NET с оператором `try/catch`, чтобы извлечь и сравнить состояние сущности, которую пользователь пытается сохранить, вместе с текущим состоянием сущности, сохраненным в базе данных:

```
try  
{  
    dbContext.Set<Product>().Add(instance);  
    dbContext.SaveChanges();  
}  
  
catch (DbUpdateConcurrencyException ex)  
{  
    var entry = ex.Entries.Single();  
    var databaseValues = (Product)entry.GetDatabaseValues().ToObject();  
    var clientValues = (Product)entry.Entity;  
    if (databaseValues.Name != clientValues.Name)  
        // Зарегистрировать в журнале исключение параллелизма.  
}  
  
catch (DataException ex)  
{  
    // Зарегистрировать в журнале исключение, связанное с данными.  
}  
  
catch (Exception ex)  
{  
    // Зарегистрировать в журнале общее исключение.  
}
```

Построение уровня доступа к данным

Выбор подхода к проектированию уровня доступа к данным является критически важным решением, которое может повлиять на оставшуюся часть приложения. Для образцового приложения Ebui был выбран подход Entity Framework Code First. Главные причины такого выбора заключались в том, что бизнес-модель Ebui построена с использованием концепций проектирования, управляемого предметной областью (<http://www.domaindrivendesign.org>), а также в том, что команда разработки желала применить подход с игнорированием постоянства, который обеспечил бы возможность поддержки в приложении множества типов моделей постоянства, включая реляционные базы данных, облачное хранилище данных и базы данных NoSQL.

Использование подхода Entity Framework Code First

Движущей силой подхода Code First является возможность применения классов POCO. В рамках подходов Database First и Model First классы модели, сгенерированные EF, унаследованы от базового класса `EntityObject`, который предоставляет необходимое оснащение для отображения класса на лежащую в основе схему базы данных. Поскольку подходы Database First и Model First требуют наследования классов хранения от класса `EntityObject`, они не поддерживают игнорирование постоянства.

Вместо использования для отображения базового сущностного класса в подходе Code First установлен набор соглашений для отображения классов POCO.

- Имена таблиц определяются с применением формы множественного числа имени сущностного класса.
- Имена столбцов выводятся из имен свойств.
- Первичные ключи основываются на свойствах с именами `ID` и `имяКлассаID`.
- Стандартная строка соединения соответствует имени класса `DataContext`.

Аннотации данных Code First

Инфраструктура Entity Framework включает множество атрибутов аннотирования данных, которые разработчики могут использовать для управления обработкой отображаемых сущностей (табл. 8.1). Обратите внимание, что в ASP.NET MVC Framework некоторые из этих атрибутов применяются для проверки достоверности на уровне полей.

Таблица 8.1. Аннотации данных Code First

Аннотация	Описание
<code>Column</code>	Имя столбца базы данных, порядковый номер и тип данных для отображения свойства
<code>ComplexType</code>	Используется на классах, которые не содержат ключей и не могут управляться инфраструктурой Entity Framework. Обычно применяется для управления скалярными свойствами в связанной сущности
<code>ConcurrencyCheck</code>	Используется для указания, должно ли свойство принимать участие в проверках оптимистического параллелизма
<code>DatabaseGenerated</code>	Используется для пометки свойства, которое должно быть сгенерировано базой данных
<code>ForeignKey</code>	Используется для идентификации связанной сущности; представляет ограничение внешнего ключа, устанавливаемое между таблицами

Аннотация	Описание
InverseProperty	Используется для идентификации свойства, которое представляет другой конец отношения
Key	Одно или более свойств, применяемых для уникальной идентификации сущности
MaxLength	Максимальная длина свойства (столбца)
MinLength	Минимальная длина свойства (столбца)
NotMapped	Помечает свойство, которое не будет отображаться инфраструктурой Entity Framework
Required	Помечает свойство как являющееся обязательным (не допускающим null)
StringLength	Определяет минимальную и максимальную длину поля
Table	Используется для определения имени таблицы, применяемого для сущности
Timestamp	Помечает свойство (столбец), содержащий метку времени, который проверяется перед сохранением изменений

Переопределение соглашений

Хотя соглашения направлены на улучшение продуктивности разработки, в Entity Framework допускается наличие ситуаций, когда необходимо нарушить одно и более применяемых соглашений, и для этого предлагается API-интерфейс, который позволяет разработчикам обходить существующие соглашения.

Ниже приведен пример сущностного класса, сконфигурированного с использованием атрибута Key, т.е. аннотации данных, которая переопределяет стандартное отображение первичного ключа:

```
public class Product
{
    [Key]
    public string MasterKey { get; set; }
    public string Name { get; set; }
}
```

В дополнение к применению атрибутов для переопределения соглашений Entity Framework разработчики могут также удалять любые стандартные соглашения или даже расширять их, создавая собственные соглашения.

В следующем коде показано, как удалить соглашение PluralizingTableNameConvention, чтобы можно было использовать для таблиц имена в единственном числе:

```
public class ProductEntities : DbContext
{
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        // Модифицировать соглашения Code First,
        // чтобы не использовалось PluralizingTableName.
        modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
    }
}
```

Бизнес-модель предметной области EBuy

Бизнес-модель образцового приложения EBuy состоит из нескольких классов РОСО, которые созданы с применением принципов проектирования, управляемого предметной областью. Каждая сущность РОСО унаследована от базового сущностного класса, который содержит общие поведения и атрибуты, разделяемые всеми классами в бизнес-модели.

Поскольку приложение EBuy спроектировано с учетом принципов SOLID (описанных в разделе “SOLID” главы 5), базовый класс Entity реализует два интерфейса: специальный интерфейс IEntity, который определяет правила именования для безопасного при использовании в URL ключевого имени сущности, и поддерживаемый в .NET Framework интерфейс IEquatable, который позволяет сравнивать друг с другом различные экземпляры одного и того же сущностного типа. Базовый класс выглядит следующим образом:

```
public interface IEntity
{
    /// <summary>
    /// Уникальный (и безопасный для использования в URL)
    /// открытый идентификатор сущности.
    /// </summary>
    /// <remarks>
    /// Идентификатор, который должен быть открыт в веб и т.д.
    /// </remarks>
    string Key { get; }

}

public abstract class Entity<TId> : IEntity, IEquatable<Entity<TId>>
where TId : struct
{
    [Key]
    public virtual TId Id
    {
        get
        {
            if (_id == null && typeof(TId) == typeof(Guid))
                _id = Guid.NewGuid();
            return _id == null ? default(TId) : (TId)_id;
        }
        protected set { _id = value; }
    }
    private object _id;

    [Unique, StringLength(50)]
    public virtual string Key
    {
        get { return _key ?? GenerateKey(); }
        protected set { _key = value; }
    }
    private string _key;

    protected virtual string GenerateKey()
    {
        return KeyGenerator.Generate();
    }
}
```

Класс, унаследованный от класса Entity, должен определять тип, используемый для его идентификатора. Обратите внимание на другие поведения, определенные в классе, такие как то, что свойство Key должно содержать уникальное значение, которое должно иметь не более 50 символов. Кроме того, класс переопределяет операции равенства для корректного сравнения множества экземпляров одного и того же объекта модели.

Модель Payment унаследована от базового класса Entity; она применяет идентификатор на основе GUID и содержит элементарные и сложные свойства. Сложные свойства используются для представления отношений с другими сущностями в модели. Например, Payment включает отношения с объектами Auction и User:

```
public class Payment : Entity<Guid>
{
    public Currency Amount { get; private set; }
    public Auction Auction { get; private set; }
    public DateTime Timestamp { get; private set; }
    public User User { get; set; }

    public Payment(User user, Auction auction, Currency amount)
    {
        User = user;
        Auction = auction;
        Amount = amount;
        Timestamp = Clock.Now;
    }

    private Payment()
    {
    }
}
```

Одна важная концепция работы с моделью предметной области связана с разделением модели на один и более контекстов, причем каждый контекст определен как агрегатный кластер, состоящий из ассоциированных объектов, которые действуют в качестве единой логической единицы. Каждый кластер содержит единственный агрегированный корень, являющийся главной сущностью, с которой связаны все остальные сущности и без которой они не могут существовать.

Агрегированный корень образцового приложения ЕВу — это класс Auction. Класс Auction представляет главную сущность в приложении, без которой все остальные классы не могут существовать.

На рис. 8.2 показаны основные классы, образующие модель предметной области ЕВу, и ассоциации между ними. Так как Auction является агрегированным корнем, он имеет отношения с другими основными сущностями, включая класс Bid, который представляет собой коллекцию предложений цены, сделанных разными пользователями для аукционного товара, класс User, представляющий две различных роли (аукционист и покупатель), и класс Payment, который представляет платеж, осуществляющий победителем торгов аукционисту.

В следующем коде демонстрируется внутреннее устройство класса Auction вместе со всеми его связанными сущностями и поведениями. Тип `ICollection<T>` используется для определения различных связанных классов, в том числе Bid, Category и Image.

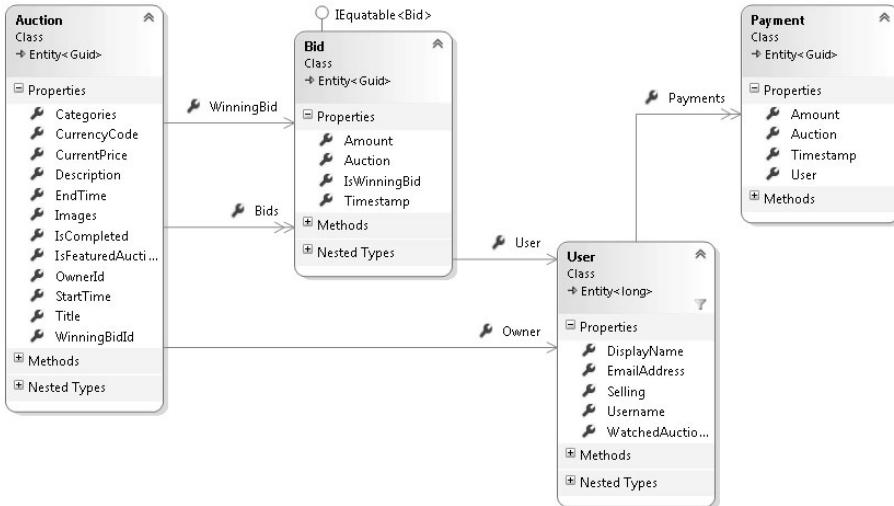


Рис. 8.2. Модель предметной области

Главное поведение класса касается размещения новых предложений цен.

```

public class Auction : Entity<Guid>
{
    public virtual string Title { get; set; }
    public virtual string Description { get; set; }
    public virtual DateTime StartTime { get; set; }
    public virtual DateTime EndTime { get; set; }
    public virtual Currency CurrentPrice { get; set; }

    public Guid? WinningBidId { get; set; }
    public virtual Bid WinningBid { get; private set; }

    public bool IsCompleted
    {
        get { return EndTime <= Clock.Now; }
    }

    public virtual bool IsFeaturedAuction { get; private set; }
    public virtual ICollection<Category> Categories { get; set; }
    public virtual ICollection<Bid> Bids { get; set; }
    public virtual ICollection<WebsiteImage> Images { get; set; }

    public long OwnerId { get; set; }
    public virtual User Owner { get; set; }

    public virtual CurrencyCode CurrencyCode
    {
        get
        {
            return (CurrentPrice != null) ? CurrentPrice.Code : null;
        }
    }

    public Auction()
    {
    }
}
  
```

```

        Bids = new Collection<Bid>();
        Categories = new Collection<Category>();
        Images = new Collection<WebsiteImage>();
    }
    public void FeatureAuction()
    {
        IsFeaturedAuction = true;
    }
    public Bid PostBid(User user, double bidAmount)
    {
        return PostBid(user, new Currency(CurrencyCode, bidAmount));
    }
    public Bid PostBid(User user, Currency bidAmount)
    {
        Contract.Requires(user != null);
        if (bidAmount.Code != CurrencyCode)
            throw new InvalidBidException(bidAmount, WinningBid);
        if (bidAmount.Value <= CurrentPrice.Value)
            throw new InvalidBidException(bidAmount, WinningBid);
        var bid = new Bid(user, this, bidAmount);
        CurrentPrice = bidAmount;
        WinningBidId = bid.Id;
        Bids.Add(bid);
        return bid;
    }
}

```

Работа с контекстом данных

Подход доступа к данным ADO.NET Entity Framework Code First требует от разработчика создать класс контекста доступа к данным, который унаследован от `DbContext`. Этот класс должен содержать свойства для каждой сущности в модели предметной области. Специальный класс контекста доступа может переопределять методы базового класса контекста для поддержки любой специализированной логики запросов и сохранения данных, а также реализовывать любую необходимую логику для отображения сущностей.

Ниже приведен контекст данных Entity Framework Code First, который содержит две сущности: `Categories` и `Products`. После определения класса контекста данных с помощью запроса LINQ извлекается список товаров специфической категории:

```

public partial class DataContext : DbContext
{
    public DbSet<Category> Categories { get; set; }
    public DbSet<Product> Products { get; set; }
}
public IList<Product> GetProductsByCategory(Category item)
{
    IList<Product> result = null;
    var db = new DataContext();
    result = db.Products.Where(q => q.Category.Equals(item)).ToList();
    return result;
}

```

Для поддержки специальных отображений сущностей, таких как отношения “многие ко многим”, метод OnModelCreating() контекста данных должен быть переопределен. Взгляните на пример настройки отношения “многие ко многим” между таблицами Bids и Auctions базы данных:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<Bid>()
        .HasRequired(x => x.Auction)
        .WithMany()
        .WillCascadeOnDelete(false);
}
```

По умолчанию Entity Framework будет искать в файле web.config веб-приложения ASP.NET MVC строку соединения, имеющую то же самое имя, что и специальный класс контекста доступа к данным:

```
<connectionStrings>
<add name="Ebuy.DataAccess.DataContext"
      providerName="System.Data.SqlClient"
      connectionString="Data Source=.\SQLEXPRESS;AttachDbFilename=
          |DataDirectory|\Ebuy.mdf;Initial Catalog=Ebuy;
          Integrated Security=True;User Instance=True;
          MultipleActiveResultSets=True"
    />
</connectionStrings>
```

Вместо работы напрямую с контектом данных Entity Framework разработчик может использовать шаблон Repository. Это устанавливает уровень абстракции между контроллерами приложения и инфраструктурой Entity Framework. Обеспечение реализации классом репозитория некоторого интерфейса (IRepository) позволяет разработчику применять контейнер IoC для внедрения репозитория в контроллер.

Ниже приведен пример репозитория, который настроен для абстрагирования от контекста данных Entity Framework:

```
public class Repository : IRepository
{
    private readonly DbContext _context;
    public Repository(DbContext context)
    {
        _context = context;
        _isSharedContext = isSharedContext;
    }
}
```

Сортировка, фильтрация и разбиение данных на страницы

При поддержке сортировки, фильтрации и разбиения данных на страницы ADO.NET Entity Framework полагается на запросы LINQ (Language Integrated Query – язык интегрированных запросов) для взаимодействия с базой данных.

Разработчики создают и вызывают запросы LINQ в отношении контекста данных Entity Framework. После определения запроса LINQ для запуска запроса вызывается один из методов LINQ, такой как `ToList()`. Инфраструктура Entity Framework преоб-

разует команды LINQ в соответствующий синтаксис SQL и выполняет запрос. После завершения работы запроса возвращенный им результат преобразуется в строго типизированную коллекцию на основе сущностного типа, определенного запросом.

Страница поиска образцового приложения EBuy (рис. 8.3) демонстрирует совместную работу этих методов для обеспечения фильтрации, сортировки и разбиения на страницы результатов поиска. Пользователи могут вводить ключевое слово для поиска, изменять свойство, по которому осуществляется сортировка, и перемещаться по страницам результатов поиска, возвращенных из базы данных; все это делается посредством простых запросов LINQ.

Search Result			
Sort By: Price Page 1 of 2			
Nintendo Wii Console Black Wii Sports Resort takes the inclusive, fun and intuitive controls of the original Wii Sports to the next level, introducing a whole new set of entertaining and physically immersive activities.	\$8.00	Closed	
Xbox 360 Elite The Xbox 360 Elite gaming system is the ultimate in gaming	\$25.00	2 days, 1 hours	
Xbox 360 Kinect Sensor with Game Bundle You are the controller with Kinect for Xbox 360!	\$26.00	2 days, 20 hours	
Sony Playstation 3 120GB Slim Console The fourth generation of hardware released for the PlayStation 3 entertainment platform, the PlayStation 3 120GB system is the next stage in the evolution of Sony's console gaming powerhouse.	\$54.00	Closed	
Burton Mayhem snow board Burton Mayhem snow board: 159cm wide	\$69.00	4 days, 3 hours	
Lock of John Lennon's hair Lock of John Lennon's hair	\$94.00	Closed	
Sony PSP Go The smallest and mightiest PSP system yet.	\$95.00	Closed	

Рис. 8.3. Страница поиска приложения EBuy

После того как пользователь вводит ключевое слово и щелкает на кнопке отправки, активизируется SearchController. Метод Index принимает параметр SearchCriteria. Этот класс содержит все ключевые поля, которые пользователь может изменять на экране поиска: ключевое слово, поле для сортировки и количество элементов для отображения на странице.

Когда этот метод вызван, связыватель модели ASP.NET MVC отобразит поля формы поиска на класс SearchCriteria. В дополнение к полям на странице поиска, этот класс содержит несколько вспомогательных методов, упрощающих получение поля сортировки и количества элементов для отображения на странице:

```
public class SearchCriteria
{
    public enum SearchFieldType
    {
        Keyword,
        Price,
        RemainingTime
    }
    public string SearchKeyword { get; set; }
    public string SortByField { get; set; }
    public string PageSize { get; set; }
    public int CurrentPage { get; set; }
```

```

public int GetPageSize()
{
    int result = 5;
    if (!string.IsNullOrEmpty(this.PagingSize))
    {
        int.TryParse(this.PagingSize, out result);
    }
    return result;
}

public SearchFieldType GetSortByField()
{
    SearchFieldType result = SearchFieldType.Keyword;
    switch (this.SortByField)
    {
        case "Price":
            result = SearchFieldType.Price;
            break;
        case "Remaining Time":
            result = SearchFieldType.RemainingTime;
            break;
        default:
            result = SearchFieldType.Keyword;
            break;
    }
    return result;
}
}

```

Представление Search (пример 8.1) использует несколько важных концепций, которые уже должны быть вам знакомы. Это представление задействует механизм событий jQuery для перехвата событий, ассоциированных с полями критерия поиска, поэтому если одно из этих полей изменяется, скрытое поле для отслеживания текущей страницы обновляется и форма отправляется контроллеру SearchController. Представление Search также применяет класс SearchViewModel, который содержит свойства для установки критерия поиска и результатов.

Пример 8.1. Представление Search

```

@model SearchViewModel
 @{
     ViewBag.Title = "Index";
 }

<script type="text/javascript">
$(function () {
    $("#SortByField").change(function () {
        $("#CurrentPage").val(0);
        SubmitForm();
    });
    $("#PageSize").change(function () {
        $("#CurrentPage").val(0);
        SubmitForm();
    });
})

```

```

$( "#Previous" ).click(function () {
    var currentPage = $( "#CurrentPage" ).val();
    if (currentPage != null && currentPage > 0) {
        currentPage--;
        $( "#CurrentPage" ).val(currentPage);
    }
    SubmitForm();
});

$( "#Next" ).click(function () {
    var currentPage = $( "#CurrentPage" ).val();
    if (currentPage) {
        currentPage++;
        $( "#CurrentPage" ).val(currentPage);
    }
    SubmitForm();
});
});

function SubmitForm() {
    document.forms["SearchForm"].submit();
}

</script>
@using (Html.BeginForm("Index", "Search", FormMethod.Post, new { id = "SearchForm" }))
{
    <div class="SearchKeyword">
        @Html.TextBoxFor(m => m.SearchKeyword, new {@class="SearchBox"})
        <input id="Search" type="submit" value=" " class="SearchButton" />
    </div>

    <h2>Search Result</h2>

    <div>
        <div class="SearchHeader">
            @Html.Hidden("CurrentPage", @Model.CurrentPage)
            <div class="PagingContainer">
                <span class="CurrentPage">Page @Model.CurrentPage of @Model.MaxPages
                </span>
                
                
                <div class="PageSize">
                    @Html.DropDownListFor(m => m.PagingSize, new SelectList(Model.PagingSizeList))
                </div>
            </div>
            <div class="SortingContainer">
                <span>Sort By:</span>
                @Html.DropDownListFor(m => m.SortByField,
                                     new SelectList(Model.SortByFieldList))
            </div>
        </div>
        <div class="SearchResultContainer">
            <table>
                @foreach (var item in @Model.SearchResult)
                {
                    var auctionUrl = Url.Auction(item);

```

```

<tr>
    <td class="searchDescription">
        <div class="fieldContainer">
            <a href="@auctionUrl">
                @Html.SmallThumbnail(@item.Image, @item.Title)</a>
            </div>
        <div class="fieldContainer">
            <div class="fieldTitle">@item.Title</div>
            <div class="fieldDescription">
                @item.Description
            </div>
        </div>
    </td>
    <td class="centered-field">@item.CurrentPrice</td>
    <td class="centered-field">@item.RemainingTimeDisplay</td>
</tr>
}
</table>
</div>
</div>
}

```

После того как пользователь вводит поисковое ключевое слово или изменяет одно из полей критерия поиска, активизируется SearchController. Действие Index обрабатывает входящие запросы и проверяет, содержит ли входной экземпляр SearchCriteria какие-либо данные критерия поиска. Если пользователь вводит ключевое слово, оно применяется для фильтрации аукционных товаров, возвращенных в результатах поиска. Метод Query() в репозитории используется для отправки данных фильтра (например, ключевого слова) контексту данных Entity Framework, который строит запрос SQL и возвращает фильтрованные данные контроллеру:

```

public ActionResult Index(SearchCriteria criteria)
{
    IQueryable<Auction> auctionsData = null;
    // Фильтровать аукционные товары посредством ключевого слова.
    if (!string.IsNullOrEmpty(criteria.SearchKeyword))
        auctionsData = _repository.Query<Auction>(
            q => q.Description.Contains(criteria.SearchKeyword));
    else
        auctionsData = _repository.All<Auction>();
    // Код для загрузки модели представления.
    return View(viewModel);
}

```

Когда пользователь изменяет поле для сортировки, активизируется SearchController. Этот контроллер проверяет экземпляр SearchCriteria, чтобы определить, по какому полю сортировать результаты поиска.

Для сортировки данных вызывается LINQ-команда OrderBy, которой передается сортирующее поле (q => q.CurrentPrice.Value):

```

switch (criteria.GetSortByField())
{
    case SearchCriteria.SearchFieldType.Price:
        auctionsData = auctionsData.OrderBy(q => q.CurrentPrice.Value);
        break;
}

```

```

        case SearchCriteria.SearchFieldType.RemainingTime:
            auctionsData = auctionsData.OrderBy(q => q.EndTime);
            break;
        case SearchCriteria.SearchFieldType.Keyword:
        default:
            auctionsData = auctionsData.OrderBy(q => q.Description);
            break;
    }
}

```

Когда пользователь щелкает на кнопке перехода к предыдущей или следующей странице либо изменяет количество элементов, отображаемых на странице, активизируется SearchController. Контроллер проверяет параметр criteria, чтобы определить, сколько страниц отображать, и получает предыдущую или следующую страницу результатов поиска. Метод PageSearchResult() затем вызывает специальный расширяющий метод Page() для выяснения, сколько аукционных товаров отображать после того, как все фильтры поиска будут применены.

Если количество аукционных товаров больше запрошенного размера страницы, возвращается результат поиска для текущей страницы. Однако если размер страницы больше или равен количеству аукционных товаров, метод возвращает все аукционные товары, которые удовлетворяют поисковому запросу:

```

private IEnumerable<Auction> PageSearchResult(SearchCriteria criteria,
                                                IQueryable <Auction> auctionsData)
{
    IEnumerable<Auction> result = null;
    var NumberOfItems = auctionsData.Count();
    if (NumberOfItems > criteria.GetPageSize())
    {
        var MaxNumberOfPages = NumberOfItems / criteria.GetPageSize();
        if (criteria.CurrentPage > MaxNumberOfPages)
            criteria.CurrentPage = MaxNumberOfPages;
        result = auctionsData.Page(criteria.CurrentPage, criteria.GetPageSize());
    }
    else
    {
        result = auctionsData.ToArray();
    }
    return result;
}

```

Для упрощения разбиения данных на страницы был создан расширяющий метод C# для типов IEnumerable<T>. Этот метод использует LINQ-команды Skip и Take для возврата только количества элементов на основе параметров текущего номера страницы и размера страницы:

```

public static class CollectionExtensions
{
    public static IEnumerable<T> Page<T>(this IEnumerable<T> source,
                                            int pageIndex, int pageSize)
    {
        // Индекс страницы не может быть отрицательным.
        Contract.Requires(pageIndex >= 0, "Page index cannot be negative");
        // Размер страницы не может быть отрицательным.
        Contract.Requires(pageSize >= 0, "Page size cannot be negative");
    }
}

```

```
        int skip = pageIndex * pageSize;
        if (skip > 0)
            source = source.Skip(skip);
        source = source.Take(pageSize);
        return source;
    }
}
```

Резюме

В этой главе были описаны распространенные шаблоны доступа к данным и показано их применение на примере ADO.NET Entity Framework. Кроме того, рассматривались различные подходы доступа к данным, поддерживаемые Entity Framework, и демонстрировалось использование подхода Code First для построения уровня доступа к данным. Также былоделено внимание применению классов POCO и шаблона Repository при разработке веб-приложения ASP.NET MVC.

Безопасность

В этой главе обсуждаются вопросы построения безопасных веб-приложений ASP.NET MVC, включая руководство по защите веб-приложений; отличия, которые должны приниматься во внимание при защите Интернет-, интранет- и экстранет-приложений; а также использование встроенной в .NET Framework функциональности, которая помогает предотвратить возникновение распространенных проблем безопасности, характерных для большинства веб-приложений.

Построение защищенных веб-приложений

Бенджамин Франклин однажды заметил, что легче предупредить ошибку, чем исправлять ее последствия. Это утверждение выражает философию, которую вы должны принять, когда дело доходит до защиты разрабатываемых веб-приложений: мир – место опасное, и веб-приложения часто представляют собой привлекательные цели для потенциальных злоумышленников, поэтому имеет смысл должным образом подготовиться.

К сожалению, в отношении безопасности веб-приложения не существует единого перечня “правильных рецептов”. Это не сводится к простому включению какой-либо библиотеки или вызову метода. Безопасность – это то, что должно быть заложено в приложение с самого начала, а не пристраиваться в последнюю минуту.

Тем не менее, существует ряд принципов безопасности, объясняемых в следующих нескольких разделах, которые могут оказать значительное влияние на создание более защищенных веб-приложений ASP.NET MVC. Если вы будете учитывать эти принципы при проектировании и реализации веб-приложений, то получите намного большие шансы избежать некоторых распространенных и серьезных ошибок безопасности.

Обеспечьте защиту в глубину

Тот факт, что веб-сайт имеет единственный уровень приложения, который напрямую взаимодействует с внешним миром, вовсе не означает, что за обеспечение безопасности отвечает только этот один уровень. В противоположность этому, защищенные системы основаны на том, что каждый уровень приложения и подсистема отвечает за собственную безопасность и должна действовать в качестве собственного привратника; часто предполагают, что определенный уровень будет вызываться только из другого, доверенного уровня, но это не всегда так! Взамен каждый уровень должен действовать так, как будто бы он всегда взаимодействует напрямую с внешним миром, аутентифицируя и авторизуя пользователей перед тем, как позволить им выполнить любое действие.

Никогда не доверяйте введенным данным

Любые данные, введенные пользователем или другой системой, должны всегда рассматриваться как потенциально опасные, поэтому обязательно нужно выполнять проверку достоверности таких данных перед их использованием. Никогда не предполагайте, что данным можно доверять, т.к. они уже проверены в другом месте!

Например, проверка достоверности формы на стороне клиента с использованием JavaScript в браузере помогает создавать более привлекательный пользовательский интерфейс, но это не должно быть единственной линией обороны, поскольку потенциальный злоумышленник вполне может отправить форму прямо на сервер и обойти любую существующую клиентскую проверку достоверности. Клиентская проверка достоверности должна рассматриваться как средство повышения удобства, но не средство безопасности, и контроллеры должны всегда выполнять проверку данных, которые они принимают.

Соблюдайте принцип наименьшего уровня привилегий

Выполняйте код, используя учетную запись с только теми привилегиями, которые нужны для текущей задачи (принцип наименьшего уровня привилегий), и проектируйте свое приложение так, чтобы не требовать повышения прав до тех пор, пока это не станет действительно необходимым. В сценариях, требующих повышенных прав, ограничивайте эти права, выдавая их на как можно более короткий промежуток времени: по завершении работы немедленно отбирайте права. Например, вместо запуска веб-сайта от имени учетной записи администратора лишь для того, чтобы разрешить доступ к диску с целью сохранения загруженных файлов, создайте учетную запись пользователя, который имеет на текущей машине доступ к специфической папке только для создания новых файлов, но не для их удаления, обновления или выполнения.

Предполагайте, что внешние системы являются незащищенными

Аутентифицировать и авторизовать компьютер или внешнее приложение не менее важно, чем конечного пользователя-человека. Когда системы нуждаются во взаимодействии друг с другом, предусмотрите использование разных системных учетных записей для каждой внешней системы, с которой взаимодействует ваше приложение, а затем ограничьте полномочия каждой учетной записи для обеспечения доступа только к тем операциям, которые необходимо выполнять внешней системе.

Сокращайте поверхность атаки

Избегайте открытия информации или операций без настоятельной необходимости. Например, контроллеры ASP.NET MVC должны минимизировать количество действий, доступ к которым они открывают, и ограничивать входные параметры этих действий только данными, которые действительно нужны для работы.



Атрибут BindAttribute привязки моделей ASP.NET MVC предоставляет свойства `Include` и `Exclude`, позволяющие указывать разделяемый запятыми список свойств модели, которые, соответственно, должны быть привязаны или проигнорированы.

Кроме того, фиксируйте в журнале и обрабатывайте любые исключения, генерируемые вашим приложением, чтобы не возвращались никакие системные подробности, касающиеся путей к файлам, имен учетных записей или схемы базы данных, которыми злоумышленник мог бы воспользоваться для выяснения сведений о системе.

Отключайте ненужные средства

Наиболее распространенными являются автоматизированные атаки, нацеленные на широко известные уязвимости в популярных платформах или службах. Чтобы не стать целью такого типа атак, имеет смысл удалить или отключить средства или службы, которые не требуются приложению.

Например, если приложение не отправляет электронную почту, необходимо отключить все средства электронной почты на размещающей машине.

Защита приложения

Веб-приложения часто имеют дело с несколькими видами пользователей. Например, приложение может взаимодействовать с конечными пользователями, входящими в основную аудиторию приложения; администраторами, которые выполняют задачи, связанные с мониторингом и развертыванием приложения; и с "пользователями" учетной записи приложения или служебной учетной записи, которые применяются для взаимодействия между различными уровнями приложения либо обращения к внешним службам.

На высоком уровне первым моментом, который понадобится учесть при проектировании веб-приложения ASP.NET, является модель аутентификации, требуемая для сайта; после этого функциональные возможности приложения можно разделить на разные роли авторизации. Перед тем как двигаться дальше, определим несколько важных терминов.

- **Аутентификация.** Аутентификация – это процесс идентификации того, кто получает доступ к приложению. Аутентификация дает ответы на следующие вопросы: кем является текущий пользователь и представляет ли этот пользователь того, за кого себя выдает? Инфраструктуры ASP.NET и ASP.NET MVC позволяют выбирать между аутентификацией *Windows* и аутентификацией с помощью форм.
- **Авторизация.** Авторизация – это процесс определения уровня прав, которые аутентифицированный пользователь должен иметь для доступа к защищенным ресурсам. Инфраструктура ASP.NET MVC позволяет декларативно добавлять атрибут `AuthorizeAttribute` к действиям контроллеров (или целым контроллерам) для программной проверки, имеет ли пользователь специфичную роль.

После определения модели безопасности, которая будет применяться приложением для конечных пользователей, самое время решить, каким образом разнообразные уровни приложения будут взаимодействовать друг с другом.

Один из самых популярных способов организации внутреннего взаимодействия предусматривает создание *служебной учетной записи приложения*, которой выдается наименьший объем привилегий, требуемых для взаимодействия уровней друг с другом.

Например, если веб-приложению необходима лишь возможность поиска и вывода данных без их модификации, служебная учетная запись предоставит только доступ по чтению в локальную файловую систему веб-сервера и базу данных приложения.

В случаях, когда требуется доступ по чтению и записи, поверхность атаки все равно может быть минимизирована за счет выдачи служебной учетной записи детализированного доступа к специфическим системам и средствам: например, ограничение загрузки до единственной папки и предоставление доступа по записи в указанные таблицы, которые приложение должно обновлять или вставлять в них данные.

Защита интранет-приложения

Веб-приложения для внутренних и внешних сетей (интранет- и экстранет-приложения) чаще всего конфигурируются для применения аутентификации Windows. При таком подходе с HTTP-запросом посыпается пользовательский маркер безопасности Windows по мере того, как пользователь перемещается по веб-приложению.

Приложение затем использует этот маркер для проверки допустимости учетной записи на локальной машине (или в домене), а также для определения ролей, к которым принадлежит пользователь, чтобы выяснить, имеет ли пользователь возможность выполнения заданного действия. Если пользователь не прошел аутентификацию или авторизацию, веб-браузер запрашивает у него удостоверение безопасности.

Настройка аутентификации Windows

Инфраструктура ASP.NET MVC упрощает создание веб-приложения, в котором применяется аутентификация Windows. Все что понадобится – создать новое приложение ASP.NET MVC 4 с использованием шаблона Intranet Application (Интранет-приложение), как показано на рис. 9.1. Этот шаблон устанавливает режим аутентификации `<authentication mode="Windows" />` в файле `web.config` приложения.

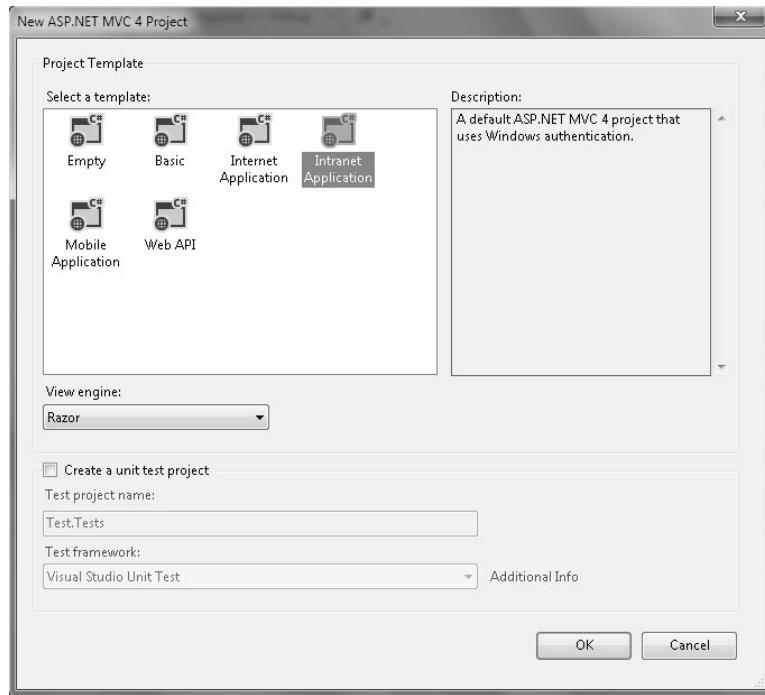


Рис. 9.1. Создание нового интранет-приложения ASP.NET MVC

Для развертывания интранет-приложения ASP.NET MVC потребуется сначала сконфигурировать аутентификацию Windows на веб-сервере, на котором будет размещаться приложение. В следующих разделах рассматривается конфигурирование веб-серверов Internet Information Server (IIS) и IIS Express.



Веб-сервер, встроенный в Visual Studio, хорошо подходит для локального развертывания, однако он обрабатывает отказы аутентификации Windows не так, как это делает IIS. Таким образом, чтобы развернуть и протестировать функциональность веб-приложения, требующего аутентификацию Windows, необходимо применять веб-сервер IIS Express или IIS 7.0+.

Конфигурирование IIS Express

Ниже перечислены шаги, которые понадобится выполнить, чтобы сконфигурировать IIS Express для размещения веб-приложения ASP.NET, требующего аутентификацию Windows.

1. В окне проводника решения (Solution Explorer) щелкните правой кнопкой мыши на веб-проекте ASP.NET и выберите в контекстном меню пункт Use IIS Express... (Использовать IIS Express...), как показано на рис. 9.2. Подтвердите выбор в открывшемся диалоговом окне IIS Express.
2. Вернитесь в окно Solution Explorer, выберите свой проект и нажмите клавишу <F4> для отображения свойств проекта. Установите свойство Anonymous Authentication (Анонимная аутентификация) в Disabled (отключена), а свойство Windows Authentication (Аутентификация Windows) – в Enabled (включена), как показано на рис. 9.3.

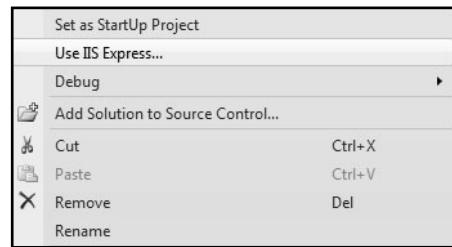


Рис. 9.2. Выбор IIS Express

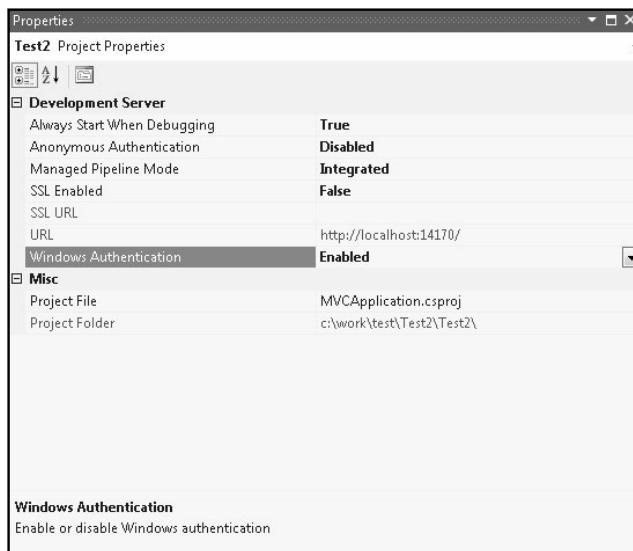


Рис. 9.3. Настройка свойств проекта

Конфигурирование IIS 7

Конфигурирование аутентификации Windows в IIS 7 несколько отличается от ее конфигурирования в IIS Express. В следующих шагах показано, как включить аутентификацию Windows в IIS 7.

1. Запустите диспетчер служб IIS 7.0, щелкните правой кнопкой мыши на одном из веб-сайтов (например, Default Web Site), как показано на рис. 9.4, и выберите в контекстном меню пункт Add Web Application (Добавить приложение).
2. В открывшемся диалоговом окне Add Application (Добавление приложения), показанном на рис. 9.5, выберите псевдоним, пул приложений (удостоверьтесь, что выбран пул приложений ASP.NET 4.0) и физический путь, который будет действовать в качестве корневой папки веб-сайта.
3. Выберите только что созданное веб-приложение и щелкните на значке Authentication (Аутентификация), после чего отключите анонимную аутентификацию и включите аутентификацию Windows (рис. 9.6).



Рис. 9.4. Выбор веб-сайта для конфигурирования аутентификации Windows

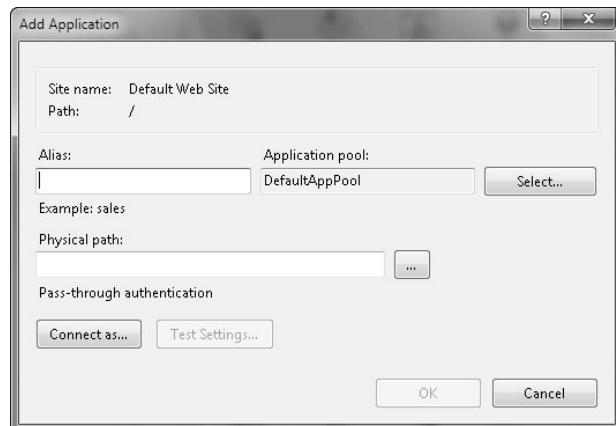


Рис. 9.5. Диалоговое окно Add Application

Authentication		
Name	Status	Response Type
Anonymous Authentication	Disabled	HTTP 401 Challenge
ASP.NET Impersonation	Disabled	HTTP 302 Login/Redirect
Basic Authentication	Disabled	HTTP 401 Challenge
Forms Authentication	Disabled	
Windows Authentication	Enabled	HTTP 401 Challenge

Рис. 9.6. Настройка параметров аутентификации

Использование атрибута **AuthorizeAttribute**

Атрибут **AuthorizeAttribute** позволяет декларативно ограничивать доступ к действию контроллера, отклоняя запросы, когда пользователь пытается обратиться к действию контроллера, к которому он не имеет прав доступа.



Если приложение размещается на встроенным веб-сервере Visual Studio, будет возвращаться пустая страница, т.к. этот веб-сервер не поддерживает отказы аутентификации Windows.

В табл. 9.1 приведен список свойств **AuthorizeAttribute**.

Таблица 9.1. Свойства **AuthorizeAttribute**

Свойство	Описание
Order	Определяет порядок выполнения фильтра действия (унаследовано от FilterAttribute)
Roles	Получает или устанавливает роли, требуемые для доступа к действию
Users	Получает или устанавливает имена пользователей, которым разрешен доступ к действию

В следующем фрагменте кода показано, как сконфигурировать действие, которое доступно только определенным пользователям. В этом примере доступ к действию контроллера **AdminProfile** разрешен только пользователям **Company\Jess** и **Company\Todd**; все остальные пользователи получат отказ:

```
[Authorize(Users = @"Company\Jess, Company\Todd")]
public ActionResult AdminProfile()
{
    return View();
}
```

Хотя в этом примере указан список явных имен пользователей, вместо этого лучше применять группы Windows. Это значительно упростит управление доступом к приложению, поскольку одно и то же имя группы Windows может использоваться во многих местах, а состав этой группы легко изменять с помощью стандартных инструментов Windows.

Чтобы использовать группы Windows вместо имен пользователей, заполните свойство **Roles** именами групп Windows:

```
[Authorize(Roles = "Admin, AllUsers")]
public ActionResult UserProfile()
{
    return View();
}

[Authorize(Roles = "Executive")]
public ActionResult ExecutiveProfile()
{
    return View();
}
```

Аутентификация с помощью форм

Ограничения подхода аутентификации Windows могут стать действительно очевидными, как только база пользователей приложения расширяется за пределы локального домена. В таких сценариях – т.е. на большинстве публично доступных сайтов в Интернете – вместо аутентификации Windows должна применяться *аутентификация с помощью форм*.

В случае подхода аутентификации с помощью форм ASP.NET выдает зашифрованный cookie-набор HTTP (или значение строки запроса, если cookie-наборы у пользователя отключены), предназначенный для идентификации аутентифицированных пользователей во всех будущих запросах. Данный cookie-набор тесно связан с пользовательским сеансом ASP.NET, поэтому если возникает тайм-аут сеанса или пользователь закрывает веб-браузер, сеанс и cookie-набор становятся недействительными, так что пользователю нужно будет войти заново для установки другого сеанса.



Настоятельно рекомендуется всегда, когда возможно, применять SSL в сочетании с аутентификацией с помощью форм. Шифрование SSL автоматически обработает конфиденциальные данные пользовательского удостоверения, которые в противном случае отправлялись бы серверу в виде чистого читабельного текста. Дополнительные сведения по настройке SSL доступны по адресу:

<http://learn.iis.net/page.aspx/144/how-to-set-up-ssl-on-iis/>

Поскольку аутентификация с помощью форм в настоящее время является наиболее распространенным подходом, большинство шаблонов веб-приложений ASP.NET MVC (все, за исключением шаблона Intranet Application) заранее сконфигурированы на использование этого вида аутентификации.

Шаблон Internet Application (Интернет-приложение) в ASP.NET MVC даже предоставляет готовую стандартную реализацию, генерируя класс контроллера по имени `AccountController` и связанные с ним представления для входа и регистрации нового пользователя.

На рис. 9.7 показана стандартная форма входа, входящая в состав шаблона Internet Application. Эта форма включает типичные средства формы входа, в том числе поля для ввода имени пользователя и пароля, флажок `Remember me?` (Запомнить меня?) и ссылку `Register` (Зарегистрироваться) для новых пользователей.

The screenshot shows a modal dialog box titled "Identification". The main instruction text reads: "Log On. Enter your username and password below." Below this are two input fields: "User name" and "Password", each with a placeholder character. To the right of the "User name" field is a "Remember me?" checkbox. At the bottom of the form is a "Log On" button, and just above it is a link "Register if you don't have an account."

Рис. 9.7. Стандартная форма входа в рамках шаблона Internet Application

Стандартная форма регистрации нового пользователя (рис. 9.8) также включаетстроенную проверку достоверности паролей, которая проверяет на предмет совпадения пароли, введенные в двух полях, и соблюдение требования о том, что они должны содержать не менее шести символов.

The screenshot shows a registration form with the following fields and errors:

- User name: User173
- Email address: User173@domain.com
- Password: (empty field)
- Confirm password: (empty field)

An error message at the top of the form states: "The password and confirmation password do not match."

At the bottom of the form is a "Register" button.

Рис. 9.8. Стандартная форма регистрации нового пользователя

Рассматривайте эти представления в качестве стартовой точки и модифицируйте их для приведения к требованиям конкретного приложения.

Контроллер AccountController

Контроллер AccountController оснащен полной поддержкой типичных сценариев аутентификации на основе форм ASP.NET. Он имеет готовую реализацию регистрации новых пользователей на сайте, аутентификации существующих пользователей и даже логики, позволяющей пользователям изменять свои пароли.

Внутренне этот контроллер использует стандартные поставщики членства и ролей ASP.NET, как сконфигурировано в файле web.config приложения. Стандартная конфигурация поставщика членства ASP.NET предусматривает хранение информации о членстве для сайта в базе данных SQL Express по имени ASPNETDB.MDF, расположенной в папке /App_Data приложения. Однако из-за того, что SQL Express в основном подходит для сценариев разработки, имеет смысл переключиться на применение хранилища данных производственного уровня, такого как полная версия Microsoft SQL Server или даже Active Directory. В качестве альтернативы можно создать и зарегистрировать собственные поставщики, чтобы расширить возможности поставщика членства для включения дополнительных полей или предоставления дополнительных средств, которые не предлагаются стандартными поставщиками.

К счастью, изменение параметров стандартного поставщика членства или даже переключение на специальный поставщик членства – дело только конфигурации. Чтобы изменить конфигурацию поставщика членства, просто обновите раздел <membership> в файле web.config приложения:

```

<membership defaultProvider="DefaultMembershipProvider">
    <providers>
        <add name="DefaultMembershipProvider"
            type="System.Web.Providers.DefaultMembershipProvider,
                System.Web.Providers, Version=1.0.0.0, Culture=neutral,
                PublicKeyToken=31bf3856ad364e35"
            connectionStringName="DefaultConnection"
            enablePasswordRetrieval="false"
            enablePasswordReset="true"
            requiresQuestionAndAnswer="false"
            requiresUniqueEmail="false"
            maxInvalidPasswordAttempts="5"
            minRequiredPasswordLength="6"
            minRequiredNonalphanumericCharacters="0"
            passwordAttemptWindow="10" applicationName="/" />
    </providers>
</membership>

```

Выполнение аутентификации пользователей

Когда пользователь пытается получить доступ к защищенному разделу веб-приложения, он перенаправляется на форму входа. Контроллер `AccountController` использует атрибут `AllowAnonymousAttribute` для указания на то, что действие `Login` является исключением из правил авторизации, и не аутентифицированные пользователи могут иметь к нему доступ. Если этого не сделать, пользователи никогда не смогут увидеть форму входа для выполнения аутентификации!

Поскольку действие контроллера `Login` доступно посредством стандартных HTML-форм входа и форм на основе AJAX, это действие просматривает свойство `content` строки запроса для определения, откуда оно было вызвано. Если значение параметра `content` отличается от `null`, действие возвращает AJAX-версию формы входа; иначе предполагается, что запрос является стандартным полностраничным запросом браузера, поэтому действие возвращает полное представление с формой входа. Код для входа пользователя выглядит примерно так:

```

[AllowAnonymous]
public ActionResult Login()
{
    return ContextDependentView();
}

private ActionResult ContextDependentView()
{
    string actionName =
        ControllerContext.RouteData.GetRequiredString("action");
    if (Request.QueryString["content"] != null)
    {
        ViewBag.FormAction = "Json" + actionName;
        return PartialView();
    }
    else
    {
        ViewBag.FormAction = actionName;
        return View();
    }
}

```

После того как пользователь введет свое удостоверение безопасности и щелкнет на кнопке отправки, форма входа отправит это удостоверение версии `HttpPost` действия `Login`, чтобы попытаться аутентифицировать пользователя.

Аутентификация пользователей с помощью форм в ASP.NET является двухшаговым процессом.

1. Сначала действие `Login` вызывает метод `Membership.ValidateUser()`, чтобы выяснить, является ли пользователь действительным.
2. Затем, если поставщик членства сообщает, что удостоверение действительно, действие вызывает метод `FormsAuthentication.SetAuthCookie()` для создания маркера безопасности пользователя.

Наконец, после того как пользователь успешно вошел, он будет либо перенаправлен на URL, где изначально требовалась аутентификация, либо, если он переходил прямо на страницу входа, возвращен на домашнюю страницу приложения. Если во время аутентификации возникла ошибка, пользователь возвращается обратно на форму входа для повторения попытки.

Ниже приведен код, который обрабатывает все это:

```
[AllowAnonymous]
[HttpPost]
public ActionResult Login(LoginModel model, string returnUrl)
{
    if (ModelState.IsValid)
    {
        if (Membership.ValidateUser(model.UserName, model.Password))
        {
            FormsAuthentication.SetAuthCookie(model.UserName, model.RememberMe);
            if (Url.IsLocalUrl(returnUrl))
            {
                return Redirect(returnUrl);
            }
            else
            {
                return RedirectToAction("Index", "Home");
            }
        }
        else
        {
            ModelState.AddModelError("", "The user name or password provided is incorrect.");
        }
    }
    // Если управление попало сюда, произошел отказ; повторно отобразить форму.
    return View(model);
}
```

Регистрация новых пользователей

Прежде чем пользователи смогут аутентифицировать себя на сайте, они сначала должны получить учетные записи. Хотя возможна ситуация, когда администраторы веб-сайта управляют пользовательскими учетными записями вручную, более распространен подход, при котором пользователям разрешено регистрировать собственные учетные записи.

В шаблоне Internet Application из ASP.NET MVC действие контроллера Register отвечает за взаимодействие с пользователем и сбор всех данных, необходимых для создания новой пользовательской учетной записи с помощью поставщика членства.

В сущности, действие Register выглядит во многом похожим на описанное ранее действие Login – атрибут AllowAnonymous позволяет пользователям получать доступ к действию, и в действии применяется специфичная для контекста логика для возврата частичного или полного представления в зависимости от того, является ли запрос запросом AJAX.

Однако вместо аутентификации пользователей форма осуществляет отправку в адрес действия Register, указывая приложению на необходимость регистрации нового пользователя посредством поставщика членства, используя его метод Membership.CreateUser().

После того как новый пользователь успешно зарегистрирован, действие применяет тот же самый метод FormsAuthentication.SetAuthCookie(), показанный в действии Login, для автоматической аутентификации нового пользователя, после чего перенаправляет его на домашнюю страницу приложения:

```
[AllowAnonymous]
public ActionResult Register()
{
    return ContextDependentView();
}

[AllowAnonymous]
[HttpPost]
public ActionResult Register(RegisterModel model)
{
    if (ModelState.IsValid)
    {
        // Попытаться зарегистрировать пользователя.
        MembershipCreateStatus createStatus;
        Membership.CreateUser(model.UserName, model.Password, model.Email,
            passwordQuestion: null,
            passwordAnswer: null,
            isApproved: true,
            providerUserKey: null,
            status: out createStatus);

        if (createStatus == MembershipCreateStatus.Success)
        {
            FormsAuthentication.SetAuthCookie(
                model.UserName, createPersistentCookie: false);
            return RedirectToAction("Index", "Home");
        }
        else
        {
            ModelState.AddModelError("", ErrorCodeToString(createStatus));
        }
    }

    // Если управление попало сюда, произошел отказ; повторно отобразить форму.
    return View(model);
}
```

Изменение паролей

Контроллер AccountController предоставляет одно дополнительное действие, которое обычно присутствует в большинстве веб-приложений с аутентификацией с помощью форм: ChangePassword.

Процесс начинается с запроса к действию ChangePassword от пользователя, который желает изменить свой пароль. Контроллер пытается найти учетную запись этого пользователя, вызвав метод Membership.GetUser() для получения экземпляра MembershipUser, который содержит информацию аутентификации данного пользователя.

Если обнаружить пользовательскую учетную запись удалось, действие ChangePassword затем вызывает метод ChangePassword() на MembershipUser, передавая ему новый пароль.

После успешного изменения пароля пользователь перенаправляется на представление ChangePasswordSuccess, подтверждающее, что все прошло гладко и пароль изменен:

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

[HttpPost]
public ActionResult ChangePassword(ChangePasswordModel model)
{
    if (ModelState.IsValid)
    {
        // В определенных сценариях отказ ChangePassword будет генерировать
        // исключение вместо возврата значения false.
        bool changePasswordSucceeded;
        try
        {
            MembershipUser currentUser =
                Membership.GetUser(User.Identity.Name, userIsOnline: true);
            changePasswordSucceeded =
                currentUser.ChangePassword(model.OldPassword, model.NewPassword);
        }
        catch (Exception)
        {
            changePasswordSucceeded = false;
        }
        if (changePasswordSucceeded)
        {
            return RedirectToAction("ChangePasswordSuccess");
        }
        else
        {
            ModelState.AddModelError("", 
                "The current password is incorrect or the new password is invalid.");
        }
    }
    // Если управление попало сюда, произошел отказ; повторно отобразить форму.
    return View(model);
}
```

Взаимодействие через AJAX

В дополнение к стандартной модели HTML-запросов GET/POST, контроллер AccountController также поддерживает вход и регистрацию пользователей через AJAX. В следующем фрагменте показаны методы AJAX для этих возможностей:

```
[AllowAnonymous]
[HttpPost]
public JsonResult JsonLogin(LoginModel model, string returnUrl)
{
    if (ModelState.IsValid)
    {
        if (Membership.ValidateUser(model.UserName, model.Password))
        {
            FormsAuthentication.SetAuthCookie(model.UserName, model.RememberMe);
            return Json(new { success = true, redirect = returnUrl });
        }
        else
        {
            ModelState.AddModelError("", "The user name or password provided is incorrect.");
        }
    }
    // Если управление попало сюда, произошел отказ; повторно отобразить форму.
    return Json(new { errors = GetErrorsFromModelState() });
}

[AllowAnonymous]
[HttpPost]
public ActionResult JsonRegister(RegisterModel model)
{
    if (ModelState.IsValid)
    {
        // Попытаться зарегистрировать пользователя.
        MembershipCreateStatus createStatus;
        Membership.CreateUser(model.UserName, model.Password, model.Email,
            passwordQuestion: null, passwordAnswer: null, isApproved: true,
            providerUserKey: null, status: out createStatus);
        if (createStatus == MembershipCreateStatus.Success)
        {
            FormsAuthentication.SetAuthCookie(model.UserName,
                createPersistentCookie: false);
            return Json(new { success = true });
        }
        else
        {
            ModelState.AddModelError("", ErrorCodeToString(createStatus));
        }
    }
    // Если управление попало сюда, произошел отказ; повторно отобразить форму.
    return Json(new { errors = GetErrorsFromModelState() });
}
```

Эти методы должны выглядеть знакомыми; единственное крупное отличие между действиями на основе AJAX и действиями на основе GET заключается в том, что контроллер возвращает ответ JSON (через JsonResult), а не ответ HTML (через ViewResult).

Авторизация пользователей

Авторизация пользователей работает одинаково как при аутентификации с помощью форма, так и при аутентификации Windows – помещение атрибута `AuthorizeAttribute` на действие контроллера ограничивает доступ к этому действию только аутентифицированными пользователями:

```
[Authorize]
public ActionResult About()
{
    return View();
}
```

Если не аутентифицированный пользователь попытается обратиться к этому действию контроллера, ASP.NET MVC отклонит запрос и перенаправит пользователя на страницу входа.

Когда подобное происходит, исходная запрошенная страница передается в качестве параметра (`ReturnUrl`) странице входа, к примеру, `/Account/LogOn?ReturnUrl=%2fProduct%2fCreateNew`. После успешного входа пользователь будет перенаправлен на исходную запрошенную страницу.

При работе с аутентификацией с помощью форм некоторые страницы, такие как домашняя страница приложения или страница контактов, могут быть доступны всем пользователям. Атрибут `AllowAnonymous` выдает права доступа не аутентифицированным пользователям:

```
[AllowAnonymous]
public ActionResult Register()
{
    return ContextDependentView();
}
```

В дополнение к декларативной авторизации пользователя посредством свойств `User` и `Groups` атрибута `AuthorizeAttribute`, можно также обращаться напрямую к вошедшему пользователю, вызывая метод `User.Identity.Name()` или `User.IsInRole()` для проверки, авторизован ли пользователь на выполнение заданного действия:

```
[HttpPost]
[Authorize]
public ActionResult Details(int id)
{
    Model model = new Model();
    if (!model.IsOwner(User.Identity.Name))
        return View("InvalidOwner");
    return View();
}
```

Предохранение против атак

Управление безопасностью пользователей – это лишь первый шаг в защите веб-приложения. Второй и более важной задачей является предохранение против атак со стороны потенциальных злоумышленников.

Хотя большинство пользователей обычно не собираются взламывать приложения, с которыми работают, они очень часто обнаруживают ошибки, приводящие к возникновению брешей в безопасности. Более того, злоумышленники выступают во многих

личинах, от простых нарушителей, которые получают удовольствие от самого процесса, до тех, кто проводит автоматизированные атаки с применением специальных "червей" и вирусов, предназначенных для воздействия на известные уязвимости.

Независимо от типа нарушителя, самым важным оружием защиты от атак является планирование и внедрение принципов безопасности, обсуждаемых в этой главе. Также очень важно организовать соответствующий мониторинг и аудит, чтобы если атака случится, операционный персонал и команда разработки могли идентифицировать ее причину и противостоять будущим атакам подобного рода.

В нескольких следующих разделах рассматриваются наиболее распространенные типы атак на веб-приложения и шаги, которые можно предпринять, чтобы защититься от них.

Внедрение SQL-кода

Атака внедрением SQL-кода происходит, когда злоумышленник обманом путем заставляет веб-приложение принять параметры, которые приводят к выполнению произвольного запроса с использованием ненадежных данных. В целях демонстрации приведенный ниже пример является очень простым. Однако имейте в виду, что злоумышленники обычно не пользуются простой логикой; наоборот, они создают сложные алгоритмы, которые идентифицируют средства атаки и затем задействуют их.

Давайте начнем с простого примера, уязвимого к атаке внедрением SQL-кода. Во всех примерах применяется схема базы данных из образцового приложения EBuy, которая включает такие таблицы, как *Auctions* и *Categories*. Между таблицами *Auctions* и *Categories* (показанными на рис. 9.9 и 9.10) установлено отношение "многие ко многим", управляемое таблицей *CategoryAuctions* (рис. 9.11).

Column Name	Data Type	Allow Nulls
Id	uniqueidentifier	<input type="checkbox"/>
Title	nvarchar(500)	<input type="checkbox"/>
Description	nvarchar(MAX)	<input type="checkbox"/>
StartTime	datetime	<input type="checkbox"/>
EndTime	datetime	<input type="checkbox"/>
CurrentPrice_Code	nvarchar(MAX)	<input checked="" type="checkbox"/>
CurrentPrice_Value	float	<input type="checkbox"/>
WinningBidId	uniqueidentifier	<input checked="" type="checkbox"/>
IsFeaturedAuction	bit	<input type="checkbox"/>
OwnerId	bigint	<input type="checkbox"/>
[Key]	nvarchar(50)	<input type="checkbox"/>
User_Id	bigint	<input checked="" type="checkbox"/>
		<input type="checkbox"/>

Рис. 9.9. Таблица Auctions приложения EBuy

Column Name	Data Type	Allow Nulls
Id	bigint	<input type="checkbox"/>
Name	nvarchar(100)	<input type="checkbox"/>
ParentId	bigint	<input checked="" type="checkbox"/>
[Key]	nvarchar(50)	<input type="checkbox"/>
		<input type="checkbox"/>

Рис. 9.10. Таблица Categories приложения EBuy

Column Name	Data Type	Allow Nulls
Category_Id	bigint	<input type="checkbox"/>
Auction_Id	uniqueidentifier	<input type="checkbox"/>

Рис. 9.11. Таблица CategoryAuctions приложения EBuy

Ниже приведен пример класса контроллера, который принимает идентификатор и запрашивает таблицу Categories:

```
public ActionResult Details(string id)
{
    var viewModel = new CategoriesViewModel();
    var sqlString = "SELECT * FROM Categories WHERE id = " + id;
    var connString = WebConfigurationManager.ConnectionStrings[
        "Ebuy.DataAccess.DataContext"].ConnectionString;
    using (var conn = new SqlConnection(connString))
    {
        var command = new SqlCommand(sqlString, conn);
        command.Connection.Open();
        IDataReader reader = command.ExecuteReader();
        while (reader.Read())
        {
            viewModel.Categories.Add(new Category { Name = reader[1].ToString() });
        }
    }
    return View(viewModel);
}
```

Когда пользователи переходят к представлению `~/category/details/1%20or%201=1` в нормальных обстоятельствах, все работает так, как ожидалось – контроллер загружает одиночную категорию на основе идентификатора, переданного в строке запроса (рис. 9.12). Тем не менее, после простого изменения строки запроса на `~/category/details/1 or 1=1` открывается брешь в безопасности. Теперь вместо отображения единственной категории возвращаются все записи в таблице Categories (рис. 9.13).

Отображение полных деталей об ошибке – установка по умолчанию в любом приложении ASP.NET – это худшее, что можно сделать, потому что при этом раскрывается стек вызовов и другая конфиденциальная информация из приложения, которая может пригодиться для поиска других брешей в приложении.

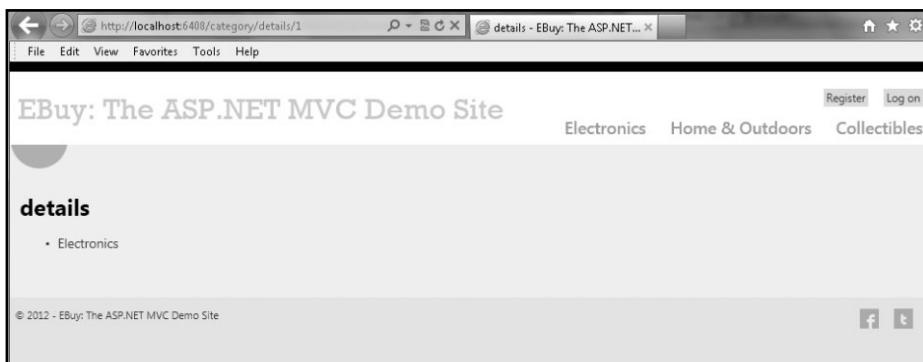


Рис. 9.12. Представление, на которое будет совершена атака внедрением SQL-кода

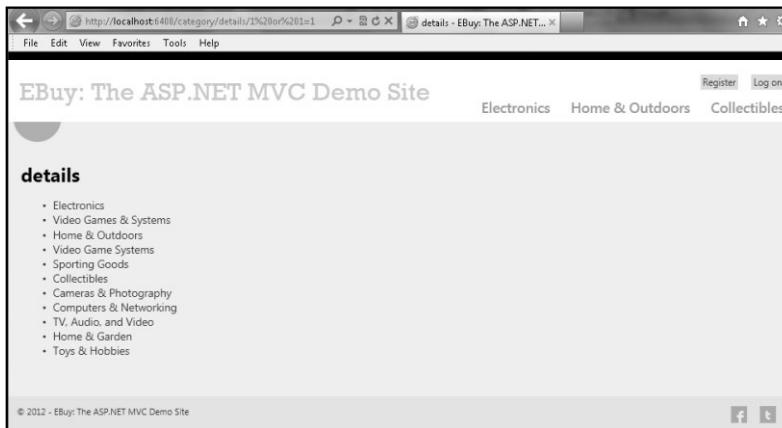


Рис. 9.13. Атака внедрением SQL-кода, использующая брешь в безопасности

Если злоумышленники успешно идентифицируют бреши вроде этой, они начинают искать что-нибудь еще, модифицируя строку запроса для использования других типов операторов SQL. На самом деле весь этот процесс может быть автоматизирован до такой степени, что злоумышленнику даже не понадобится находиться за клавиатурой.

Ниже приведены примеры действительно опасных операторов SQL, которые позволяют злоумышленнику не только выяснить, какие еще таблицы используются в приложении, но также и модифицировать их содержимое.

Первое, что злоумышленник попытается выяснить – это какие поля существуют в отображаемой таблице. Это очень полезные сведения, т.к. они позволяют узнать внешние ключи, которыми можно воспользоваться для извлечения данных из других таблиц. В этом примере злоумышленник отправляет запрос, который включает строку CategoryName: ~/category/details/1 or categoryname='". Поскольку таблица Categories такой строки не имеет, база данных генерирует исключение (рис. 9.14).

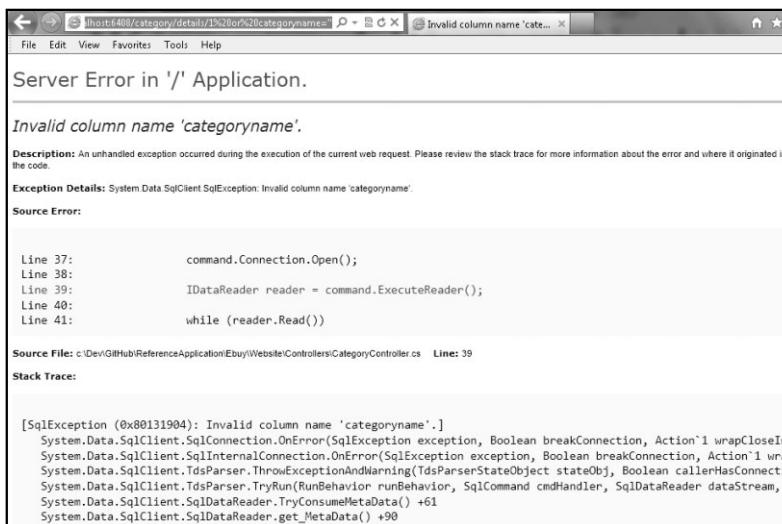


Рис. 9.14. Атака внедрением SQL-кода с несуществующим полем

Существует еще одна последовательность запросов, которые злоумышленник может отправить веб-приложению, подвергающемуся атаке. Первый запрос применяется для выяснения, какие другие таблицы существуют. После того как стало известно, что таблица существует, злоумышленник может попробовать вставить, обновить или даже удалить данные из базы приложения, как это делается во втором запросе:

```
1 OR 1=(SELECT COUNT(1) FROM Auctions)
1; INSERT INTO Auctions VALUES (1, "Test", "Description")
```

К счастью, доступно много технологий противодействия атакам за счет внедрения SQL-кода, так что вы вовсе не беззащитны от этой разновидности угроз. Лучший способ — трактовать весь ввод как опасный. Это касается данных из строк запросов, HTML-форм, заголовков запросов и любого другого ввода, поступающего в систему.

На высоком уровне существуют два подхода, которые можно применить для проверки достоверности входных данных.

- **Черные списки.** Подход на основе черных списков предусматривает проверку достоверности входных данных с использованием списка известных значений, предназначенных для исключения. Черные списки часто выглядят неплохой отправной точкой, поскольку они активно противостоят известным угрозам. Тем не менее, эффективность подхода на основе черных списков зависит от предоставляемых им данных, и они часто могут давать ложное чувство защищенности, т.к. потенциально вредоносный ввод изменяется, и новые значения пройти незаметными.
- **Белые списки.** Подход на основе белых списков следует противоположному мышлению, когда входные данные считаются допустимыми, только если они совпадают со значениями,ключенными в список известных значений. Другими словами, белые списки блокируют все значения за исключением явно разрешенных. Это делает подход на основе белых списков более безопасным, обеспечивая более тесный контроль над данными, которые разрешено передавать внутрь системы.

Хотя подходы с белыми и черными списками требуют сопровождения значений, которые они разрешают или запрещают, возможные последствия от появления вредоносного ввода (опасность, связанная с подходом черных списков) обычно намного тяжелее последствий от блокирования ввода, который должен быть расценен как допустимый, однако не является явно разрешенным (как в подходе с белыми списками).

Чтобы увидеть подход с белыми списками в действии, взгляните на следующий код, в котором приведен пример проверки того, что переданный входной параметр является числовым значением:

```
var message = "";
var positiveIntRegex = new Regex(@"^0*[1-9][0-9]*$");
if (!positiveIntRegex.IsMatch(id))
{
    message = "An invalid Category ID has been specified.";
}
```



Хотя объектно-реляционные отображатели, такие как LINQ to SQL и Entity Framework, автоматически заботятся о многих проблемах внедрения SQL-кода, эти инфраструктуры не способны защищать абсолютно от всего.

Ниже представлено руководство от Microsoft (<http://msdn.microsoft.com/ru-ru/library/cc716760.aspx>) по предотвращению атак внедрением SQL-кода для Entity Framework.

Атаки внедрением кода Entity SQL

Атаки внедрением SQL-кода могут осуществляться в Entity SQL за счет предоставления вредоносного ввода для значений, используемых в предикате запроса и в именах параметров. Во избежание риска атаки внедрением SQL-кода никогда не объединяйте пользовательский ввод с текстом команды Entity SQL.

Запросы Entity SQL принимают параметры везде, где допускаются литералы. Вы должны использовать параметризованные запросы вместо внедрения литералов из внешнего агента прямо в запрос. Рекомендуется также применять методы построителя запросов для безопасного конструирования команд Entity SQL.

Атаки внедрением кода LINQ to Entities

Хотя в LINQ to Entities возможно комбинирование запросов, это осуществляется через API-интерфейс объектной модели. В отличие от запросов Entity SQL, запросы LINQ to Entities не формируются за счет манипулирования или конкатенации строк, поэтому они не восприимчивы к обычным атакам внедрением SQL-кода.

Межсайтовые сценарии

Подобно атакам внедрением SQL-кода, *атаки межсайтовыми сценариями* (cross-site scripting – XSS) представляют собой серьезную угрозу веб-приложениям, принимающим ввод от пользователей. Основная причина успешности атак подобного типа связана с недостаточной проверкой достоверности введенных данных. Атаки XSS обычно происходят, когда злоумышленник получает возможность обманом вынудить пользователя просматривать поддельные страницы, которые с виду похожи на целевое веб-приложение, или применяет в выглядящих безобидно почтовых сообщениях встроенные ссылки, перемещающие пользователя на неожиданное местоположение. Веб-приложения, которые содержат конфиденциальные данные, весьма восприимчивы к атакам XSS. Злоумышленники часто стараются похитить cookie-наборы, которые могут содержать учетные данные для входа либо идентификаторы сессий, поскольку с их помощью они могут попытаться получить доступ к информации пользователя либо вынудить сделать его что-то опасное, вроде отправки лишнего HTML-контента или вредоносного кода JavaScript.

К счастью, в Microsoft осознают опасность межсайтовых сценариев и предлагают встроенную прямо в инфраструктуру базовую защиту в форме *проверки достоверности запросов*. Когда инфраструктура ASP.NET принимает запрос, она ищет в нем разметку или сценарии, отправляющие запрос (такие как значения полей формы, заголовки, cookie-наборы и т.д.). Если обнаружен подозрительный контент, ASP.NET отклоняет запрос, генерируя исключение. В добавок в состав ASP.NET 4.5 включена популярная библиотека XSS от Microsoft.

В некоторых сценариях приложения, подобные системам управления контентом (content management system – CMS), форумам и блогам, должны поддерживать ввод HTML-контента. Для этого в ASP.NET 4.5 введена возможность использования отложенной (или “ленивой”) проверки достоверности запросов и предлагаются методы

для доступа к непроверенным данным запросов. Однако важно применять эти средства с осторожностью и помнить, что в таком случае вы сами отвечаете за проверку достоверности ввода.

Чтобы сконфигурировать ASP.NET на использование отложенной проверки достоверности запросов, измените значение атрибута `httpRuntime:requestValidationMode` в файле `web.config` на 4.5:

```
<httpRuntime requestValidationMode="4.5" />
```

Когда отложенная проверка достоверности запросов включена, процесс проверки будет запускаться в первый раз, когда приложение обращается к коллекции запросов (например, `Request.Form["post_content"]`). Чтобы пропустить проверку достоверности ввода, используйте для доступа к непроверенной коллекции вспомогательный метод `HttpRequest.Unvalidated()`:

```
using System.Web.Helpers;
var data = HttpContext.Request.Unvalidated().Form["post_content"];
```

В Microsoft решили включить в состав ASP.NET 4.5 часть популярной библиотеки Microsoft Anti-XSS Library. Функциональные возможности шифрования реализуются классом `AntiXSSEncoded` из пространства имен `System.Web.Security.AntiXss`. С этой библиотекой можно работать напрямую, вызывая один из статических методов шифрования класса `AntiXSSEncoded`.

Простой способ применения новой функциональности противодействия атакам XSS предусматривает настройку веб-приложения ASP.NET на использование этого класса по умолчанию. Это делается установкой `encoderType` внутри файла `web.config` в `AntiXssEncoded`. В результате для шифрования всего вывода будет автоматически применяться новая функциональность XSS:

```
<httpRuntime ...
  encoderType="System.Web.Security.AntiXss.AntiXssEncoder,
  System.Web, Version=4.0.0.0, Culture=neutral,
  PublicKeyToken=b03f5f7f11d50a3a" />
```

Ниже перечислены новые средства библиотеки Microsoft Anti-XSS Library, включенные в ASP.NET 4.5:

- `HtmlEncode`, `HtmlFormUrlEncode` и `HtmlAttributeEncode`;
- `XmlAttributeEncode` и `XmlEncode`;
- `UrlEncode` и `UrlPathEncode` (нововведение ASP.NET 4.5);
- `CssEncode`.

Подделка межсайтовых запросов

Веб-сеть не является безопасным местом, и независимо от того, насколько защищенными вы пытаетесь сделать свои приложения, всегда найдется кто-то, кто попробует обойти установленные вами ограничения. Хотя атаки межсайтовыми сценариями и внедрением SQL-кода требуют большого внимания, существует другая – потенциально более серьезная – проблема, которую многие разработчики упускают из виду: подделка межсайтовых запросов (Cross-Site Request Forgery – CSRF).

Подделка межсайтовых запросов представляет потенциальный риск для безопасности по той причине, что она использует особенности работы веб-сети. Ниже приведен пример контроллера, восприимчивого к атаке CSRF. Все выглядит просто и довольно безобидно, однако этот контроллер является главной мишенью атаки CSRF:

```

public class ProductController : Controller
{
    public ViewResult Details()
    {
        return View();
    }

    public ViewResult Update()
    {
        Product product = DbContext.GetProduct();
        product.ProductId = Request.Form["ProductId"];
        product.Name = Request.Form["Name"];
        SaveUProduct(product);

        return View();
    }
}

```

В этом случае все, что понадобится предпринять находчивому злоумышленнику – создать страницу, нацеленную на контроллер. После того как злоумышленник вынудит пользователя посетить такую страницу, она попытается выполнить отправку в адрес контроллера:

```

<body onload="document.getElementById('productForm').submit()">
    <form id="productForm"
          action="http://.../Product/Update"
          method="post">
        <input name="ProductId" value="123456" />
        <input name="Name" value="My Awesome Hack" />
    </form>
</body>

```

Если пользователь уже прошел аутентификацию Windows или с помощью форм, контроллер оставит без внимания атаку CSRF. Каким же образом противостоять этому потенциально серьезному риску в плане безопасности?

Существуют два основных способа блокирования атаки CSRF.

- Ссылающийся домен.** Проверьте, имеет ли входящий запрос заголовок Referer, ссылающийся на ваш домен. Это поможет предотвратить поступление запросов, отправленных из внешних независимых источников. Такой подход имеет пару недостатков: пользователь может запретить отправку заголовка Referer по причинам конфиденциальности, а злоумышленники могут подменить этот заголовок, если у пользователя установлена устаревшая версия Adobe Flash.
- Сгенерированный пользователем маркер.** Сохраните в скрытом поле HTML-формы специфичный для пользователя маркер (например, сгенерированный вашим сервером) и удостоверьтесь, что отправленный маркер является действительным. Сгенерированный маркер может быть сохранен в сеансе пользователя или в cookie-наборе HTTP.

Использование ASP.NET MVC для противодействия атакам CSRF

Инфраструктура ASP.NET MVC включает набор вспомогательных методов, позволяющих обнаруживать и блокировать атаки CSRF за счет создания специфичного для пользователя маркера, который передается между представлением и контроллером и проверяется в каждом запросе. Для этого понадобится всего лишь вызвать вспомо-

гательный метод HTML по имени `@Html.AntiForgeryToken()`, чтобы добавить на страницу скрытое поле HTML-формы, которое будет проверяться контроллером при каждом запросе. Для усиления защиты этот вспомогательный метод HTML также принимает начальный ключ, позволяющий увеличить рандомизацию генерируемого маркера:

```
@Html.AntiForgeryToken()  
@Html.AntiForgeryToken("somerandomsalt")
```

Чтобы такой подход к противодействию атакам CSRF работал, действие контроллера, которое обрабатывает отправку формы, должно знать, что форма содержит упомянутый маркер. Для этого понадобится применить к нему атрибут `ValidateAntiForgeryTokenAttribute`. Данный атрибут обеспечит проверку наличия во входящем запросе cookie-набора и поля формы по имени `RequestVerificationToken`, а также удостоверится, что их значения совпадают:

```
[ValidateAntiForgeryToken]  
public ViewResult Update()  
{  
}
```

Вспомогательные методы противодействия атакам CSRF, входящие в состав ASP.NET MVC, очень удобны, однако с ними связано несколько ограничений, которые следует иметь в виду при их использовании.

- Легитимные пользователи должны принимать cookie-наборы. Если пользователь отключил cookie-наборы в своем веб-браузере, фильтр `ValidateAntiForgeryTokenAttribute` будет отклонять запросы этого пользователя.
- Данный метод работает только с запросами POST, но не GET. На самом деле это не имеет большого значения, т.к. запросы GET должны использоваться только для операций чтения.
- При наличии в домене любых брешей для атак XSS злоумышленник может легко получить доступ и прочитать маркер противодействия атакам CSRF.
- Готовые инфраструктуры, подобные jQuery, не передают автоматически обязательный cookie-набор и скрытое поле HTML-формы при выполнении запросов AJAX. Вам придется построить собственное решение для передачи и чтения маркера противодействия атакам CSRF.

Резюме

В этой главе было показано, как строить защищенные веб-приложения ASP.NET MVC. Вы узнали об отличиях между использованием аутентификации Windows и аутентификации с помощью форм, научились применять атрибут `AuthorizeAttribute` для авторизации различных пользователей и групп, ознакомились со способами защиты от атак внедрением SQL-кода и межсайтовыми сценариями, а также выяснили, как работать со вспомогательными методами противодействия атакам CSRF.

Разработка веб-приложений для мобильных устройств

Мобильная веб-сеть является мощным носителем, предназначенным для доставки контента огромному количеству пользователей. С увеличением числа смартфонов и последующим бурным ростом аудитории пользователей мобильной веб-сети, существенно возрастает важность включения мобильных устройств в фазы начального планирования и определения требований к разрабатываемым проектам.

Наиболее болезненный аспект разработки для мобильной веб-сети связан с тем, что не все мобильные устройства выглядят одинаково. Разные устройства обладают различными аппаратными возможностями, разрешающими способностями экранов, браузерами, функциональной поддержкой, наличием сенсорного ввода – и это далеко не полный перечень. Адаптация веб-сайта для доставки согласованного интерфейса всем мобильным устройствам является нетривиальной задачей.

В этой главе будет показано, как использовать средства ASP.NET MVC Framework – особенно новые средства, появившиеся в ASP.NET MVC 4 – для доставки насыщенного и согласованного интерфейса на максимальное количество устройств, а также каким образом корректно обрабатывать сценарии, в которых это невозможно.

Мобильные возможности ASP.NET MVC 4

Начиная с версии 3, в ASP.NET MVC Framework предлагается набор функциональных возможностей, которые способствуют упрощению разработки веб-приложений для мобильных устройств. В версии 4 эти возможности были дополнительно расширены.

В следующем списке приведены краткие описания каждой возможности мобильной разработки, появившейся в ASP.NET MVC 4. Оставшаяся часть главы посвящена применению этих новых средств в приложениях.

Шаблоны мобильных приложений ASP.NET MVC 4

Если вы хотите создать чистое мобильное веб-приложение с нуля, инфраструктура ASP.NET MVC 4 включает шаблон *Mobile Application* (Мобильное приложение), который позволяет сразу же приступить к разработке мобильного приложения. Как и в случае обычных шаблонов веб-приложений MVC, версия 4 автоматически добавляет шаблонный код для визуализации мобильных представлений, устанавливает NuGet-пакеты jQuery Mobile и создает скелетное приложение, с которого можно начать. В разделе “Шаблон Mobile Application в ASP.NET MVC 4” далее в главе приводится подробное описание нового шаблона Mobile Application в ASP.NET MVC 4.

Режимы отображения

Чтобы облегчить ориентацию на различные устройства, инфраструктура ASP.NET MVC 4 Framework поддерживает *режимы отображения* – средство, которое помогает обнаруживать и удовлетворять нуждам разных устройств.

Разные мобильные устройства имеют разную разрешающую способность, отличающуюся поведение браузера и даже могут пользоваться разными функциональными средствами веб-приложения. Вместо подгонки всех возможных вариаций устройств под единственное представление можно изолировать разные поведения и средства в отдельных представлениях, специфичных для устройств.

Например, предположим, что имеется обычное настольное представление под названием `Index.cshtml`, из которого необходимо создать несколько специфичных мобильных вариаций, таких как представление для смартфонов и еще одно представление для планшетов. Используя режимы отображения, можно создать специфичные для устройств представления, скажем, `Index.iPhone.cshtml` и `Index.iPad.cshtml`, и зарегистрировать их с помощью класса `DisplayModeProvider` инфраструктуры ASP.NET MVC 4 Framework при начальном запуске приложения. На основе вашего критерия фильтрации ASP.NET MVC Framework может автоматически искать представления, которые содержат один из этих суффиксов (`iPhone` или `iPad`), и визуализировать их вместо обычного настольного представления. (Обратите внимание, что для переключения представлений инфраструктура ASP.NET MVC следует простому соглашению об именовании файлов [*ИмяПредставления*]. [Устройство]. [Расширение].) В разделе “Представления, специфичные для браузера” далее в главе будет показано, как можно использовать это средство для обслуживания разных устройств.

Переопределение обычных представлений мобильными представлениями

В ASP.NET MVC 4 появился простой механизм, который позволяет переопределить любое представление (включая компоновки и частичные представления) для любого специфичного браузера, в том числе и мобильного. Чтобы построить мобильное представление, просто создается файл представления с суффиксом `.Mobile` в имени файла. Например, для создания мобильного представления `Index` скопируйте `Views\Home\Index.cshtml` в `Views\Home\Index.Mobile.cshtml`, после чего ASP.NET будет автоматически визуализировать в мобильном браузере это представление вместо настольного. Интересно отметить, что хотя режимы отображения позволяют специально ориентироваться на конкретный мобильный браузер, это средство обеспечивает аналогичную функциональность на более общем уровне. Оно полезно, когда представление является достаточно общим для работы в различных мобильных браузерах, или если используется инфраструктура типа jQuery Mobile, которая предоставляет согласованный интерфейс для большинства мобильных платформ.

Инфраструктура jQuery Mobile

Инфраструктура jQuery Mobile Framework переносит все богатство и функциональность пользовательских интерфейсов jQuery в мобильные приложения. Вместо того чтобы иметь дело с несовместимостями браузеров для различных устройств, можно создать единственное приложение, которое работает на всех современных мобильных устройствах. Эта инфраструктура обеспечивает все преимущества технологий прогрессивного улучшения и предоставляет гибкий дизайн интерфейса, при котором устаревшие устройства все равно могут видеть функциональное (но не настолько насыщенное) приложение, в то время как более новые устройства получают в свое распоряжение всю развитую интерактивность, предлагаемую новейшими средствами.

ми HTML 5. Инфраструктура jQuery Mobile Framework также обладает великолепной поддержкой оформления темами, которая позволяет очень легко создать узнаваемый сайт с насыщенным пользовательским интерфейсом, не принося в жертву преимущества прогрессивного улучшения. В этой главе вы увидите, что jQuery Mobile Framework позволяет перенести ваше приложение на новый качественный уровень.

Добавление мобильных возможностей в приложение

Тема разработки веб-приложений для мобильных устройств весьма обширна и включает множество аспектов, которые разработчики веб-сайтов должны учитывать при их создании. Пожалуй, наиболее важный вопрос связан с тем, как лучше всего предоставить информацию конечным пользователям и каким образом взаимодействовать с ними.

Рассмотрим настольный веб-интерфейс, в условиях которого веб-браузер имеет крупный экран, доступ в веб-сеть отличается высокой скоростью и надежностью, а пользователи могут взаимодействовать с приложениями, применяя клавиатуру и мышь. С этим резко контрастирует мобильный веб-интерфейс, который часто ограничен небольшим экраном и характеризуется прерывистым доступом в веб-сеть и наличием для ввода данных только пера или нескольких пальцев.

Эти ограничения неизбежно приводят к избирательному контенту и сокращенному набору средств по сравнению с веб-приложениями, ориентированными на настольный браузер. Тем не менее, мобильная веб-сеть также предоставляет возможности, которые в основном не доступны в настольных веб-приложениях, такие как данные, специфичные для местоположения, коммуникации на ходу и взаимодействие посредством видео и голоса.

Понимание потребностей целевой аудитории является первым шагом в формулировании мобильной стратегии. Например, взгляните на следующие распространенные примеры использования мобильных устройств.

- Попытка получения электронной почты во время прогулки по улице.
- Попытка чтения последних новостей во время езды в метро или поезде.
- Попытка проверки баланса на банковском счету, держа в одной руке чашку кофе, а в другой телефон.

Все перечисленные сценарии обладают одной общей характеристикой – внимание пользователя не сконцентрировано на единственной задаче.

В отношении веб-сайта это означает, что он должен быть сосредоточен на выдаче пользователю контента, который был бы легко и быстро воспринят и относился к текущей выполняемой задаче.

Создание мобильного представления для аукционных товаров

Разработку для мобильной веб-сети можно начать либо с добавления мобильных представлений в существующее приложение, либо создать новое мобильное приложение с нуля. На выбор того или иного способа могут влиять многие факторы, и оба подхода характеризуются своими достоинствами и недостатками, касающимися процесса разработки. С учетом этого, инфраструктура ASP.NET MVC 4 предлагает инструменты для поддержки обоих рабочих потоков, как будет показано далее в этой главе.

В данном разделе мы начнем с добавления мобильного представления к существующему настольному представлению и затем постепенно расширим это мобильное представление новыми возможностями, доступными в ASP.NET MVC 4.

Первым делом, создайте копию представления `Auctions.cshtml` и назовите ее `Auctions.Mobile.cshtml`, указав, что это мобильное представление. Чтобы ввести отличительную особенность при визуализации мобильного представления, давайте также изменим в нем заголовок <H1> на `Mobile Auctions`.

Проверить все это можно, запустив приложение и перейдя на страницу аукционных товаров в мобильном браузере. Результат показан на рис. 10.1. Заголовок страницы выглядит как `Mobile Auctions`, что подтверждает визуализацию мобильного представления (страница аукционных товаров в обычном браузере имеет заголовок `Auctions`). Средство режимов отображения инфраструктуры способно определить клиентский браузер и загрузить подходящее представление.

Инфраструктура ASP.NET MVC не только автоматически загружает “мобильные” представления, когда запрос поступает от мобильного устройства; на самом деле это распространяется также на компоновки и частичные представления – факт, который используется пакетом `jQuery.Mobile.MVC` для создания основанных на `jQuery Mobile` компоновок, оптимизированных для мобильных устройств.

Начало работы с `jQuery Mobile`

Инфраструктура `jQuery Mobile` позволяет быстро улучшить существующее представление, создав более естественный для мобильных устройств внешний вид. Наряду с этим, она поддерживает оформление темами приложения, а технология прогрессивных улучшений гарантирует, что более старые браузеры получат сокращенную (и не так хорошо выглядящую), однако все-таки функциональную и пригодную к использованию страницу.

Чтобы работать с `jQuery Mobile`, установите пакет `jQuery.Mobile.MVC` из галереи пакетов NuGet (рис. 10.2).

Этот пакет добавляет следующие файлы.

- **Инфраструктура `jQuery Mobile Framework`.** Набор файлов JavaScript (`jQuery.mobile-1.1.0.js`) и CSS (`jQuery.mobile-1.1.0.css`) вместе с их минимальными версиями и поддерживающими образами.
- **/Content/Site.Mobile.css.** Новая таблица стилей, специфичная для мобильных устройств.

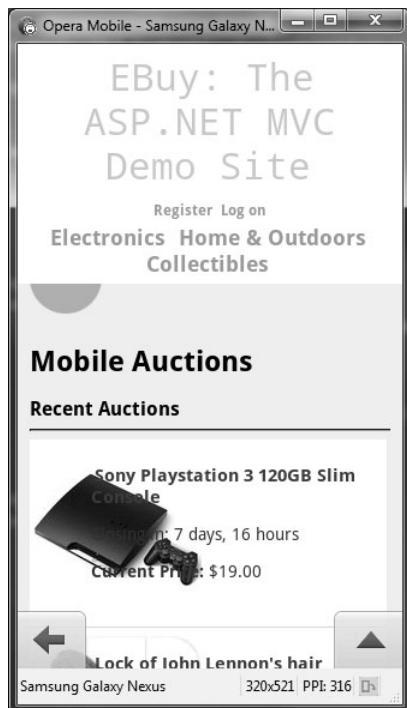


Рис. 10.1. Инфраструктура ASP.NET MVC Framework может обнаруживать и визуализировать мобильные представления автоматически

- **Views/Shared/_Layout.Mobile.cshtml.** Компоновка, оптимизированная для мобильных устройств, на которую ссылаются файлы jQuery Mobile Framework (JavaScript и CSS). Инфраструктура ASP.NET MVC будет загружать эту компоновку для мобильных представлений.
- **Компонент view-switcher.** Состоит из частичного представления Views/Shared/_ViewSwitcher.cshtml и контроллера View-SwitcherController.cs. Этот компонент отображает в мобильных браузерах ссылку, которая позволяет пользователям переключаться на настольную версию страницы. Мы рассмотрим его работу в разделе “Переключение между настольным и мобильным представлениями” далее в главе.

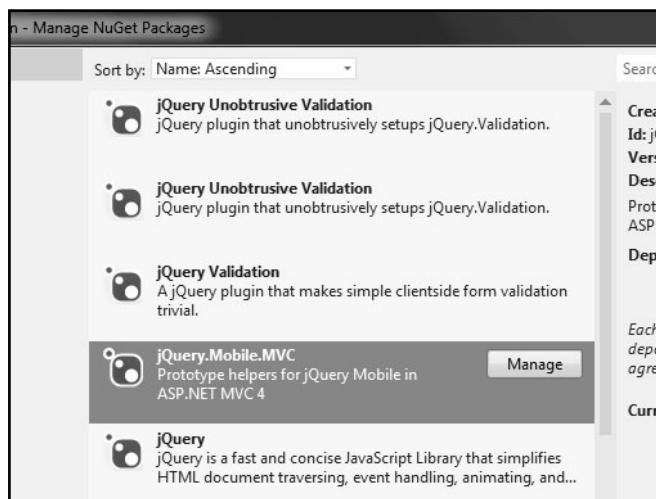


Рис. 10.2. Добавление jQuery Mobile Framework через NuGet



Инфраструктура jQuery Mobile находится в постоянной разработке, так что вы можете иметь дело с новой версией файлов.

Чтобы позволить jQuery Mobile Framework соответствующим образом стилизовать страницы, откройте файл Views/Shared/_Layout.Mobile.cshtml и модифицируйте его содержимое, как показано в следующем фрагменте:

```
<body>
    <div data-role="page" data-theme="b">
        <header data-role="header">
            <h1>@Html.ActionLink("EBuy: The ASP.NET MVC Demo Site", "Index", "Home")
            </h1>
        </header>
        <div id="body" data-role="content">
            @RenderBody()
        </div>
    </div>
</body>
```

После этого измените Auctions.Mobile.cshtml для его оптимизации для мобильной компоновки:

```

@model IEnumerable<AuctionViewModel>
<link href="@Url.Content("~/Content/product.css")"
      rel="stylesheet" type="text/css" />
 @{
     ViewBag.Title = "Auctions";
 }
 <header>
    <h3>Mobile Auctions</h3>
 </header>
 <ul id="auctions">
    @foreach (var auction in Model)
    {
        <li>
            @Html.Partial("_AuctionTile", auction);
        </li>
    }
 </ul>

```

Завершив внесение изменений, скомпилируйте и запустите приложение, а затем перейдите на домашнюю страницу приложения, используя мобильный браузер. Вы должны увидеть примерно то, что показано на рис. 10.3.

Как видите, представление *Auctions* изменилось с целью адаптации к мобильному браузеру. Хотя внешний вид получился не идеальным, пакет *jQuery.Mobile.MVC* обеспечивает основу, на которой можно быстро и легко строить мобильные представления.

Улучшение представления с помощью *jQuery Mobile*

Пакет *jQuery.Mobile.MVC* выполняет немало черновой работы, но пользовательский интерфейс по-прежнему не выглядит похожим на нормальное мобильное приложение. Тем не менее, *jQuery Mobile* предлагает множество компонентов и стилей, чтобы сделать внешний вид приложения максимально близким к настоящему мобильному приложению.

Улучшение списка аукционных товаров с помощью компонента *listview* инфраструктуры *jQuery Mobile*

Давайте начнем с улучшения списка аукционных товаров за счет использования компонента *listview* из *jQuery Mobile*. При выполнении большинства мобильных трансформаций инфраструктура *jQuery Mobile* работает с атрибутами *data-role*, поэтому для визуализации аукционных товаров в виде списка добавьте в дескриптор ** атрибут *data-role="listview"*:

```

<ul id="auctions" data-role="listview">
    @foreach (var auction in Model.Auctions)
    {

```

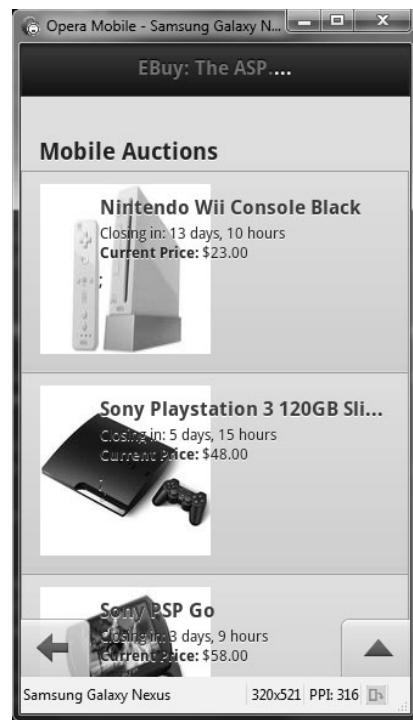


Рис. 10.3. Приложение *EBuy*, оптимизированное для мобильной компоновки

```

<li>
    @Html.Partial("_AuctionTileMobile", auction);
</li>
}
</ul>

```

Затем модифицируйте частичное представление `_AuctionTileMobile` следующим образом:

```

@model AuctionViewModel
{
    var auctionUrl = Url.Auction(Model);
}
<a href="@auctionUrl">
    @Html.Thumbnail(Model.Image, Model.Title)
    <h3>@Model.Title</h3>
    <p>
        <span>Closing in: </span>
        <span class="time-remaining" title="@Model.EndTimeDisplay">
            @Model.RemainingTimeDisplay
        </span>
    </p>
    <p>
        <strong>Current Price: </strong>
        <span class="current-bid-amount">
            @Model.CurrentPrice
        </span>
        <span class="current-bidder">
            @Model.WinningBidUsername
        </span>
    </p>
</a>

```

Переход к представлению `Auctions` в мобильном браузере теперь выдает намного более симпатичное представление, которое показано на рис. 10.4. С учетом того, что `` уже является элементом списка, вы можете посчитать избыточным добавление атрибута `data-role="listview"`. Дескриптор `` будет отображать список, но область ссылки может оказаться слишком малой, чтобы щелкать на ней на мобильном устройстве с небольшим экраном. В действительности `data-role="listview"` упрощает выполнение щелчков на элементах списка, отображая более крупную область ссылки.

Добавление к списку аукционных товаров возможности поиска с помощью атрибута `data-filter` инфраструктуры jQuery Mobile

Теперь давайте сделаем представление немноголибо более дружественным, добавив удобное поле поиска, которое позволит пользователям быстро фильтровать список аукционных товаров.

Инфраструктура jQuery Mobile позволяет сделать это очень легко — просто добавьте к дескриптору `` атрибут `data-filter="true"`:



Рис. 10.4. Визуализация дескриптора `` для аукционных товаров в виде списка с помощью jQuery Mobile

```

<ul id="auctions" data-role="listview" data-filter="true">
    @foreach (var auction in Model.Auctions)
    {
        <li class="listitem">
            @Html.Partial("_AuctionTileMobile", auction);
        </li>
    }
</ul>

```

Обновите окно мобильного браузера и обратите внимание на текстовое поле поиска в верхней части (рис. 10.5).

Попробуйте ввести что-нибудь в текстовом поле поиска, и вы увидите, что jQuery Mobile автоматически отфильтрует список для отображения только элементов, соответствующих введенному тексту (рис. 10.6).

Теперь вы знаете, что jQuery Mobile существенно упрощает трансформацию любой страницы к виду и поведению мобильного представления. В дополнение к этим возможностям, jQuery Mobile содержит множество других удобных компонентов, которые можно применять для того, чтобы сделать любое представление более доступным мобильным пользователям.

Для просмотра полного списка таких атрибутов обратитесь в документацию по адресу <http://jquerymobile.com/test/docs/api/data-attributes.html>.

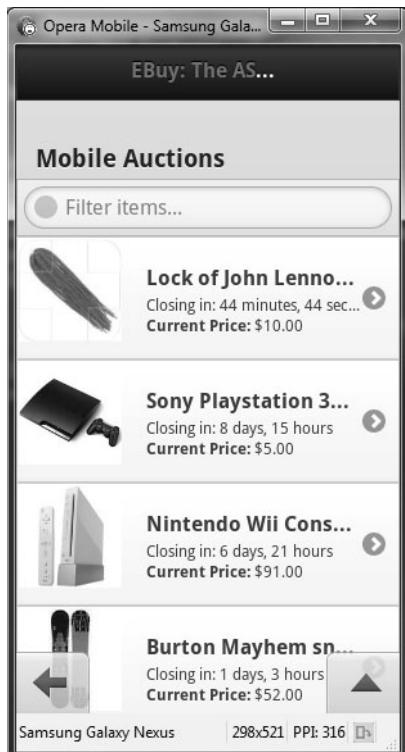


Рис. 10.5. Добавление к списку аукционных товаров возможности поиска с помощью jQuery Mobile



Рис. 10.6. Инфраструктура jQuery Mobile автоматически фильтрует список на основе введенного текста поиска

Переключение между настольным и мобильным представлениями

Всякий раз, когда предоставляется мобильная версия веб-сайта, обычно имеет смысл автоматически направлять мобильных пользователей на мобильный сайт, а также давать им возможность при желании переключаться на полный сайт.

Обратите внимание, что мобильное представление, которое включено в стандартный шаблон Mobile Application инфраструктуры ASP.NET MVC, отображает ссылку, позволяющую пользователям переключаться на “настольное представление”. За это отвечает виджет ViewSwitcher, который установлен как часть NuGet-пакета jQuery.Mobile.MVC. Чтобы разобраться, как работает этот виджет, давайте рассмотрим его компоненты. В новом частичном представлении _ViewSwitcher.cshtml находится следующая разметка:

```
@if (Request.Browser.IsMobileDevice && Request.HttpMethod == "GET")
{
    <div class="view-switcher ui-bar-a">
        @if (ViewContext.HttpContext.GetOverriddenBrowser().IsMobileDevice)
        {
            @: Displaying mobile view
            @Html.ActionLink("Desktop view", "SwitchView", "ViewSwitcher",
                new { mobile = false, returnUrl = Request.Url.PathAndQuery },
                new { rel = "external" })
        }
        else
        {
            @: Displaying desktop view
            @Html.ActionLink("Mobile view", "SwitchView", "ViewSwitcher",
                new { mobile = true, returnUrl = Request.Url.PathAndQuery },
                new { rel = "external" })
        }
    </div>
}
```

Метод GetOverriddenBrowser() возвращает объект HttpBrowserCapabilities со списком возможностей переопределенного браузера или действительного браузера, если он не переопределен, который позволяет выяснить, является ли запрашивающее устройство мобильным. Затем виджет проверяет, какое представление визуализируется — настольное или мобильное, и добавляет соответствующую ссылку для переключения между настольным и мобильным представлениями.

В качестве дополнения виджет также устанавливает свойство mobile в словаре RouteValue для указания, какое представление является активным — мобильное или настольное.

Теперь рассмотрим класс ViewSwitcherController, который содержит логику, выполняющую действие переключения:

```
public class ViewSwitcherController : Controller
{
    public RedirectResult SwitchView(bool mobile, string returnUrl)
    {
        if (Request.Browser.IsMobileDevice == mobile)
            HttpContext.ClearOverriddenBrowser();
        else
            HttpContext.SetOverriddenBrowser(mobile ? BrowserOverride.Mobile
                : BrowserOverride.Desktop);

        return Redirect(returnUrl);
    }
}
```

В зависимости от того, поступает ли запрос от мобильного устройства (как отражено свойством `Request.Browser.IsMobileDevice`), контроллер использует методы `ClearOverriddenBrowser()` и `SetOverriddenBrowser()` для сообщения ASP.NET MVC о том, каким образом обработать запрос: как мобильный браузер, отображающий мобильную версию сайта, или как настольный браузер, отображающий полную версию сайта. Добавьте перед закрывающим дескриптором `</body>` в `Layout.mobile.cshtml` следующий фрагмент, предназначенный для визуализации частичного представления `ViewSwitcher` в виде нижнего колонтитула (рис. 10.7):

```
<div data-role="footer">
    @Html.Partial("_ViewSwitcher")
</div>
```

Щелчок на ссылке `Desktop view` (Настольное представление) приводит к отображению нормального настольного представления `Auctions`. Однако обратите внимание, что настольное представление не имеет ссылки для переключения на мобильное представление. Чтобы исправить это, откройте разделяемое представление `_Layout.cshtml` и добавьте следующую строку кода:

```
@Html.Partial("_ViewSwitcher")
```

Запустите приложение и перейдите на любую страницу в мобильном браузере – вы увидите, что виджет переключателя представлений отображает ссылки для визуализации мобильного представления и полного настольного представления (рис. 10.8).

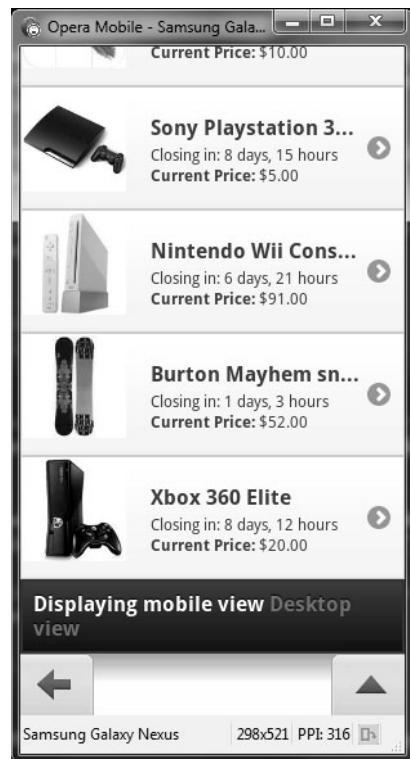


Рис. 10.7. Переключатель представлений в нижнем колонтитуле страницы

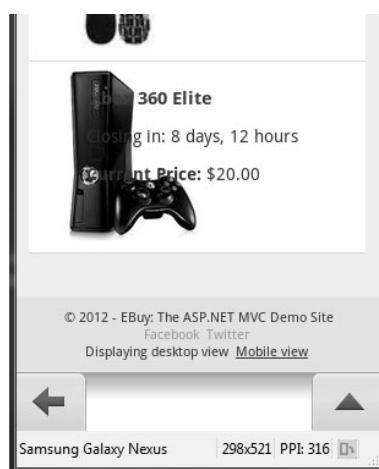


Рис. 10.8. Переключатель представлений в настольном представлении

Устранение настольных представлений на мобильном сайте

Вы заметите, что при отсутствии мобильного представления инфраструктура ASP.NET MVC визуализирует настольное представление в мобильной компоновке.

Соблюдение разметки, основанной на стандартах, помогает в отображении до некоторой степени полезного представления, но могут быть случаи, когда нужно просто отключить такую возможность.

Чтобы сделать это, установите `RequireConsistentDisplayMode` в `true`:

```
@{  
    Layout = "~/Views/Shared/_Layout.cshtml";  
    DisplayModeProvider.Instance.RequireConsistentDisplayMode = true;  
}
```

В результате визуализация любого стандартного (немобильного) представления внутри мобильной компоновки будет запрещена. Это можно также сделать глобально для всех представлений, установив свойство `RequireConsistentDisplayMode` в `true` внутри файла `/Views/_ViewStart.cshtml`.

Совершенствование мобильного интерфейса

Мобильные браузеры способны отображать HTML-страницы с разной степенью качества. Однако нельзя полагаться на один лишь браузер в предоставлении удобного пользовательского интерфейса, поскольку браузеры могут оперировать только на общем уровне установки размеров страниц и изображений. Только вы, как автор контента, можете решать, какие элементы являются наиболее релевантными, подсвечивая на небольшом экране одни элементы и отбрасывая другие. Следовательно, именно на вас возлагается ответственность за то, чтобы сайт выглядел аккуратно и был функциональным в разных браузерах.

К счастью, для совершенствования отображения сайта можно использовать такие технологии, как адаптивная визуализация и прогрессивные улучшения, при этом ASP.NET MVC 4 и jQuery Mobile позволяют делать это очень легко. Мы рассмотрим все аспекты в следующих разделах.

Адаптивная визуализация

Адаптивная визуализация — это технология, которая позволяет представлению “приспосабливаться” под возможности браузера. Например, пусть имеется набор вкладок на странице, причем при щелчке каждая вкладка осуществляет вызов AJAX для извлечения отображаемого на ней контента. Если поддержка JavaScript отключена, вкладка, как правило, будет не в состоянии отобразить любой контент. Однако за счет использования адаптивной визуализации вкладка просто укажет на URL с контентом, так что пользователь все равно сможет увидеть этот контент.

Другим примером может служить панель навигации, отображающая горизонтальный список ссылок. Хотя она выглядит хорошо в настольном представлении, эта панель может переполнить небольшой экран мобильного устройства. Используя адаптивную визуализацию, панель навигации можно представить в виде раскрывающегося списка с аналогичной функциональностью на устройствах с малыми экранами. Преимущество от применения этой технологии заключается в том, что она позволяет представить “функциональный” или “пригодный к использованию” сайт различным браузерам и устройствам с разными возможностями. Уровень обслуживания может варьироваться на основе возможностей устройства, но все же сайт остается работоспособным.

Помните, что если вы хотите, чтобы посетители возвращались на сайт снова и снова, то должны удостовериться в эффективности пользовательского интерфейса приложения независимо от устройства, на котором оно запускается.

Инфраструктура ASP.NET MVC 4 включает такие адаптивные технологии в основном через jQuery Mobile Framework.

Дескриптор окна просмотра

В компьютерной графике окно просмотра означает прямоугольную область просмотра. Применительно к браузерам это окно браузера, в котором отображается HTML-документ. Другими словами, это воображаемая конструкция, которая содержит дескриптор `<html>`, в свою очередь, являющийся корневым элементом для всей разметки.

Что произойдет при увеличении или уменьшении масштаба окна браузера? А что случится, если поменяется ориентация устройства – приведет ли это к изменению окна просмотра?

На мобильных устройствах ответы на такие вопросы несколько затруднены, поскольку в действительности существует не одно, а два окна просмотра – “компоновочное” окно просмотра и “визуальное” окно просмотра.

“Компоновочное” окно просмотра никогда не изменяется; это воображаемая конструкция, которая ограничивает HTML-контент страницы. При изменении масштаба или ориентации меняется “визуальное” окно просмотра, и это влияет на то, что именно является видимым в рамках границ экрана устройства.

Вы должны рассматривать роль окна просмотра как способ предоставления функционального и удобного интерфейса конечным пользователям. Когда страница визуализируется на мобильном устройстве, важно, чтобы ее ширина не была слишком большой или слишком малой, а хорошо умещалась на экран. И когда она умещается, страница не должна выглядеть как миниатюрная, ужатая версия полной страницы; вместо этого она должна быть читабельным представлением действительной страницы.

В современных браузерах за настройку размеров визуального окна просмотра отвечают не стили CSS, а мета-дескриптор `viewport`.

Размеры визуального окна могут быть установлены следующим образом:

```
<meta name="viewport" content="width=device-width" />
```

Применяемое здесь значение `"width=device-width"` – это специальное значение, которое обеспечивает установку ширины окна просмотра равной действительной ширине устройства. Это наиболее гибкое и часто используемое значение.

Свойство `content` можно также установить в фиксированное значение, если контент лучше адаптирован для этого:

```
<meta name="viewport" content="width=320px" />
```

Теперь, независимо от того, насколько широким является экран устройства, контент будет всегда отображаться с шириной 320 пикселей, т.е. на крупных экранах пользователю может понадобиться увеличить масштаб, а на небольших экранах – уменьшить его.



Дескриптор `<meta name="viewport">` является де-факто промышленным стандартом, но пока еще формально не входит в стандарт W3C. Это средство было впервые реализовано в веб-браузере iPhone и очень скоро из-за огромной популярности iPhone стало поддерживаться всеми другими производителями.

Определение мобильных функциональных возможностей

Поскольку каждое мобильное устройство поддерживает разный набор функциональных возможностей, никогда нельзя с уверенностью предполагать, что какая-то конкретная возможность будет доступна во всех браузерах.

Например, пусть приложение использует веб-хранилище Web Storage из HTML 5, которое поддерживают многие смартфоны (такие как iPhone, Android, Blackberry и устройства Windows Phone), однако далеко не все.

Традиционно для выяснения того, может ли приложение быть запущено в определенном браузере, разработчики рассчитывают на технологии, подобные обнаружению браузеров. Вместо проверки, поддерживается ли Web Storage, классический подход заключается в проверке, является ли целевым браузером Opera Mini.

Тем не менее, с этим подходом связано несколько крупных проблем, не последними из которых являются:

- потенциальная возможность исключения браузеров, которые не были включены явно, но поддерживают это средство;
- вероятность того, что сайт не будет функционировать должным образом, если пользователь зайдет на него из другого устройства.

Ниже приведен пример применения данного подхода:

```
// Предупреждение: не используйте этот код!
if (document.all) {
    // Internet Explorer 4+
    document.write('<link rel="stylesheet" type="text/css" src="style-ie.css">');
}
else if (document.layers) {
    // Navigator 4
    document.write('<link rel="stylesheet" type="text/css" src="style-nn.css">');
}
```

Обратите внимание, что в этом примере предоставляются только таблицы стилей для браузеров Internet Explorer и Netscape Navigator 4; к тому же, в браузере должна быть включена поддержка JavaScript. Это означает, что другие браузеры, такие как Netscape 6, Netscape 7, CompuServe 7, Mozilla и Opera, могут не иметь возможности корректного просмотра сайта.

Даже если добавить явную поддержку для большинства браузеров, все равно можно упустить из виду новую версию какого-то браузера, которая уже предлагает поддержку искомого средства.

Еще одна потенциальная проблема связана с неверной идентификацией браузера.

Так как обнаружение браузера в значительной степени связано с принятием решения на основе строки агента и определенных свойств, вполне возможна неправильная идентификация конкретного браузера:

```
// Предупреждение: не используйте этот код!
if (document.all) {
    // Internet Explorer 4+
    elm = document.all['menu'];
}
else {
    // Предположить Navigator 4
    elm = document.layers['menu'];
}
```

Как видите, в предыдущем примере предполагается, что любой браузер, отличный от Internet Explorer – это Navigator 4, и предпринимается попытка использовать слои.

Это является распространенным источником проблем, когда применяются браузеры на основе Gecko и Opera.

По всем этим причинам обычно всегда явно проверять существование функциональной возможности, а не предполагать, что набор известных версий браузеров поддерживает или не поддерживает ее.

Ниже представлен тот же пример, что и ранее, но переделанный с использованием обнаружения возможностей, а не обнаружения браузеров:

```
// Если средство localStorage присутствует, использовать его.  
if ('localStorage' in window) && window.localStorage !== null) {  
    // Простой API-интерфейс свойств объектов.  
    localStorage.wishlist = '["Unicorn", "Narwhal", "Deathbear"]';  
} else {  
    // В отсутствие sessionStorage мы будем использовать cookie-набор  
    // из далекого будущего с неудобным API-интерфейсом document.cookie.  
    var date = new Date();  
    date.setTime(date.getTime()+(365*24*60*60*1000));  
    var expires = date.toGMTString();  
    var cookiestr = 'wishlist=[\"Unicorn\", \"Narwhal\", \"Deathbear\"];' +  
        ' expires='+expires+'; path=/';  
    document.cookie = cookiestr;  
}
```

Этот подход не только намного более надежен, но он также устойчив к будущим изменениям – любой браузер, в который добавляется поддержка Web Storage, автоматически получит новые функциональные возможности.

Медиа-запросы CSS

Медиа-запросы CSS (CSS Media Queries) – это технология прогрессивных улучшений, которая позволяет адаптировать или отображать альтернативные стили на основе различных условий, связанных с браузерами.

Версия 2 спецификации CSS (CSS2) разрешает указывать стили на основе типа носителя, такого как `screen` и `print`. Версия 3 спецификации CSS (CSS3) предоставляет концепцию *медиа-запросов* – технологию, которая расширяет эту концепцию, чтобы помочь обнаруживать функциональные возможности браузера стандартным путем.



К сожалению, спецификация CSS3 все еще находится на этапе рекомендации, а это означает, что медиа-запросы – и другие новые средства версии 3 спецификации CSS – не обязательно поддерживаются всеми браузерами. Поэтому важно предусмотреть стандартные стили для тех браузеров, которые не поддерживают эти средства.

Вы уже видели, что с помощью дескриптора `viewport` можно определять стандартную ширину на основе размера устройства. Хотя окно просмотра обеспечивает приемлемый внешний вид для страницы при стандартном уровне масштабирования, оно ничем не поможет, когда пользователь увеличивает или уменьшает масштаб на устройстве.

При изменении ширины компоновки нужен способ сообщения браузеру о необходимости ограничения контента до определенной ширины, чтобы он отображался правильно во всех случаях.

Давайте рассмотрим простой пример того, как это можно сделать с помощью медиа-запроса CSS:

```
body {background-color:blue;}  
@media only screen and (max-width: 800px) {  
    body {background-color:red;}  
}
```

Так как правила CSS оцениваются сверху вниз, мы начинаем с указания общего правила, касающегося того, что цвет фона тела будет синим.

Затем задается специфичное для устройства правило, переопределяющее для фона красный цвет на устройствах с шириной экрана меньше 800 пикселей.

На устройствах, где поддерживаются медиа-запросы CSS3, а ширина меньше 800 пикселей, фон будет отображаться красным цветом; иначе он будет синим. (Обратите внимание, что изменение цвета фона при увеличении или уменьшении масштаба – это не то, что обычно делается в реальном приложении; просто приведенный пример сконцентрирован на демонстрации использования медиа-запроса CSS для применения разных стилей на основе определенных условий.)

Очень важно начинать с общего правила и затем расширять это правило с помощью поддержки медиа-запросов и обнаружения функциональных возможностей.

Это позволит вашему сайту представлять насыщенный интерфейс в браузерах, которые поддерживают новейшие возможности, и в то же время визуализировать удобное отображение в более старых браузерах.

Представления, специфичные для браузера

Новое средство режимов отображения в ASP.NET MVC 4 позволяет загружать разные представления на основе предварительно определенных условий. Простым примером применения этого средства может быть создание отдельных представлений для смартфонов, имеющих меньшие экраны, и для планшетов, экраны которых больше, чем у мобильных устройств, но меньше, чем у настольных компьютеров. Создание разных представлений для таких классов устройств позволяет обеспечить оптимальное использование экранного пространства и предоставляет эффективный и насыщенный пользовательский интерфейс, настроенный с учетом возможностей устройства.

Первым делом, зарегистрируйте режимы отображения при начальном запуске приложения:

```
using System.Web.WebPages;  
  
// Зарегистрировать представления, специфичные для iPhone.  
DisplayModeProvider.Instance.Modes.Insert(0, new DefaultDisplayMode("iPhone")  
{  
    ContextCondition = (ctx => ctx.Request.UserAgent.IndexOf(  
        "iPhone", StringComparison.OrdinalIgnoreCase) >= 0)  
});  
  
// Зарегистрировать представления, специфичные для Windows Phone.  
DisplayModeProvider.Instance.Modes.Insert(0, new DefaultDisplayMode("WindowsPhone")  
{  
    ContextCondition = (ctx => ctx.Request.UserAgent.IndexOf(  
        "Windows Phone", StringComparison.OrdinalIgnoreCase) >= 0)  
});
```

Теперь создайте представление, специфичное для iPhone, скопировав файл Auctions.mobile.cshtml и переименовав его в Auctions.iPhone.cshtml.

Затем измените заголовок на iPhone Auctions, чтобы отличать его от других мобильных представлений. Запустите приложение, используя эмулятор мобильного браузера (для примеров, приведенных в главе, применялось дополнение User Agent Switcher (Переключатель агентов пользователя) для Firefox (<https://addons.mozilla.org/en-US/firefox/addon/user-agent-switcher/>), эмулирующее браузер iPhone), чтобы посмотреть на него в действии (рис. 10.9).

Чтобы построить версию страницы для Windows Phone, создайте копию файла Auctions.mobile.cshtml и переименуйте ее в Auctions.WindowsPhone.cshtml. Затем измените заголовок на Windows Phone Auctions, чтобы отличать это представление от других мобильных представлений. Запустите приложение, используя эмулятор мобильного браузера, чтобы увидеть его в действии (рис. 10.10).

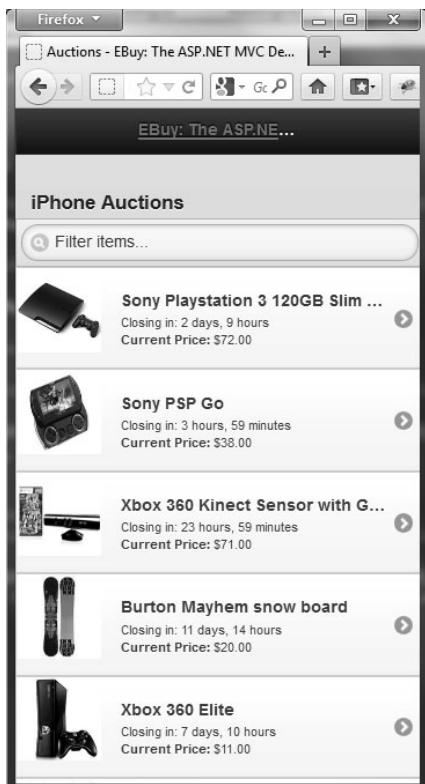


Рис. 10.9. Представление, специфичное для iPhone

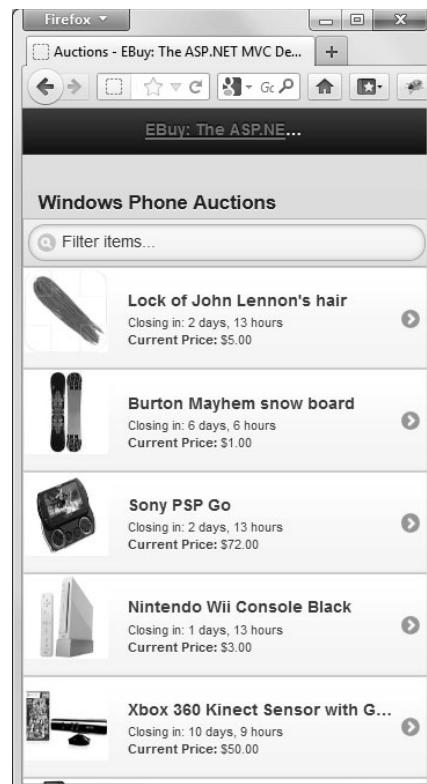


Рис. 10.10. Представление, специфичное для Windows Phone



Для выяснения, поступает ли запрос от мобильного устройства, ASP.NET внутренне проверяет его с применением предварительно определенного набора определений мобильных браузеров.

Множество сведений о функциональных возможностях браузера предоставляет класс `HttpBrowserCapabilities` (<http://msdn.microsoft.com/en-us/library/system.web.httpbrowsecapabilities.aspx>), который доступен через свойство `Request.Browser`.

Или же вместо того, чтобы полагаться на встроенные определения браузеров, можно воспользоваться службой наподобие 51Degrees.mobi (<http://51degrees.mobi/Support/Blogs/tabid/212/EntryId/26/51Degrees-mobi-and-MVC4.aspx>), которая поддерживает актуальный банк информации о разнообразных мобильных устройствах.

Создание нового мобильного приложения с нуля

Инфраструктура ASP.NET MVC 4 упрощает добавление мобильных представлений в существующее приложение, но не менее легко также создавать мобильное приложение с нуля. Это важно в случае, если отсутствует приложение, с которого можно начать, или по какой-либо причине вы не хотите смешивать мобильный и настольный сайты.

В состав ASP.NET MVC 4 входит шаблон Mobile Application, который позволяет быстро приступить к разработке мобильного приложения. Большая часть функциональности этого шаблона основана на jQuery Mobile, поэтому для построения эффективного приложения сначала необходимо освоить jQuery Mobile.

Сдвиг парадигмы в jQuery Mobile

Возможно, наиболее важным отличием при работе с jQuery Mobile является понятие “страницы”. При традиционной веб-разработке под страницей понимается одиночный HTML-документ, страница .aspx в ASP.NET Web Forms или представление .cshtml в ASP.NET MVC. Эти файлы содержат разметку и логику для визуализации одиночной страницы в браузере.

Тем не менее, в инфраструктуре jQuery Mobile Framework один файл может содержать множество мобильных “страниц”. Формально страница jQuery Mobile – это в действительности просто дескриптор `<div>` с атрибутом `data-role="page"`. В единственный файл можно поместить сколько угодно страниц, и jQuery превратит их во множество страниц, отображаемых по одной.

Так как одно обычное настольное представление может приводить к небольшим порциям множества представлений на мобильном устройстве (в основном из-за переработки страницы, чтобы сделать ее подходящей для мобильной навигации), этот подход помогает сократить количество мелких файлов, которые иначе были бы созданы для небольших порций настольного представления.

Шаблон Mobile Application в ASP.NET MVC 4

Создание нового мобильного веб-приложения начинается таким же способом, что и в случае любого другого веб-приложения ASP.NET MVC: выбор пункта меню `File⇒New⇒Project` (Файл⇒Новый⇒Проект) и затем указание типа проекта `ASP.NET MVC 4 Web Application` (Веб-приложение ASP.NET MVC 4), как видно на рис. 10.11.

На следующем экране (рис. 10.12) выберите шаблон `Mobile Application` (Мобильное приложение).

Это приведет к созданию нового приложения ASP.NET MVC с примерами контроллеров и представлений, которые наглядно демонстрируют мобильные возможности ASP.NET MVC и помогают быстро приступить к работе.

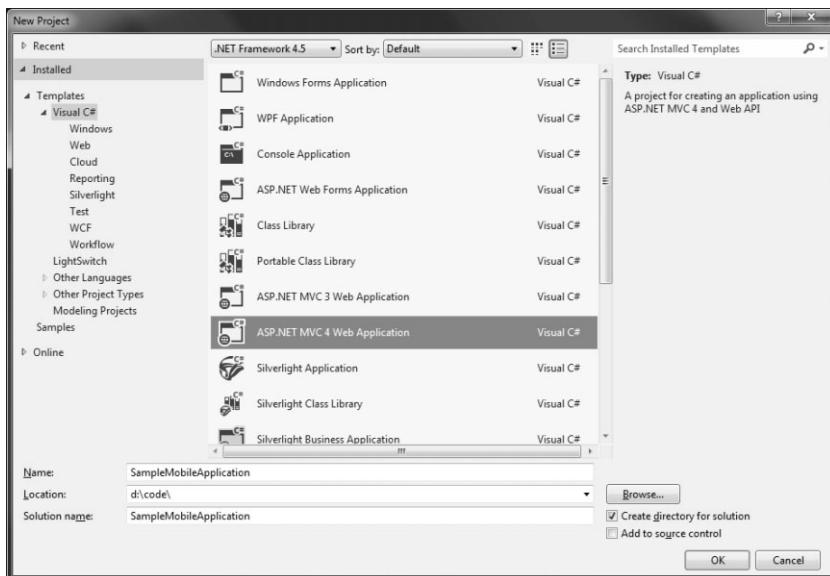


Рис. 10.11. Создание нового проекта

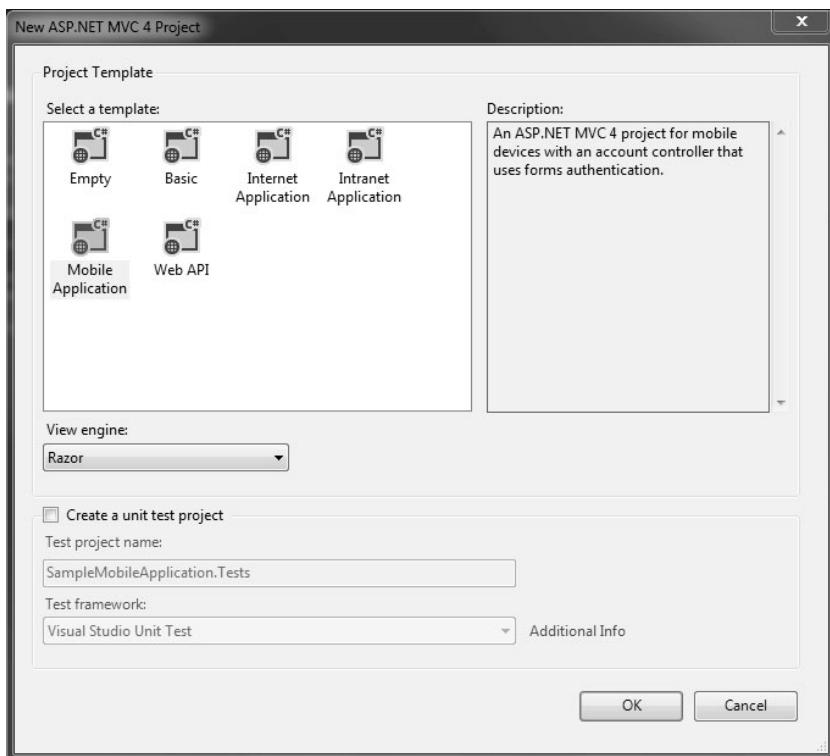


Рис. 10.12. Выбор шаблона Mobile Application

Запустите проект, нажав клавишу <F5> или выбрав пункт меню Debug⇒Start (Отладка⇒Запуск). Решение скомпилируется, после чего запустится экземпляр браузера, указывающий на мобильную домашнюю страницу веб-сайта (рис. 10.13).

Использование шаблона Mobile Application инфраструктуры ASP.NET MVC 4

Как видите, автоматически был построен большой объем шаблонного кода. Структура проекта очень похожа на структуру обычного веб-сайта, но с несколькими дополнениями.

- Папка Content теперь включает таблицы стилей для jQuery Mobile, как показано на рис. 10.14:
 - jquery.mobile-1.1.0.css (и ее минимальная версия);
 - jquery.mobile.structure-1.1.0.css (и ее минимальная версия).
- Папка Scripts содержит два новых файла, как показано на рис. 10.15:
 - jquery.mobile-1.1.0.js;
 - jquery.mobile-1.1.0.min.js.

Эти новые файлы являются частью jQuery Mobile Framework – JavaScript-инфраструктуры, которая привносит все ценные характеристики jQuery и jQuery UI в мобильные устройства.

А теперь взглянем на модифицированный файл _Layout.cshtml. Дескриптор head содержит несколько новых строк.

Мета-дескриптор viewport задает размер окна просмотра. Это важно, т.к. хотя большинство браузеров позволяют пользователям увеличивать или уменьшать масштаб, как им заблагорассудится, установка начальной ширины для контента обеспечивает лучший пользовательский интерфейс.

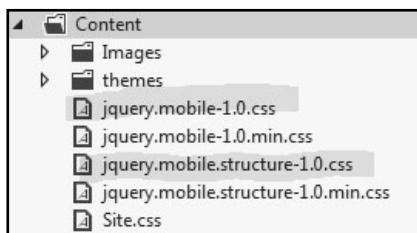


Рис. 10.14. Папка Content нового проекта

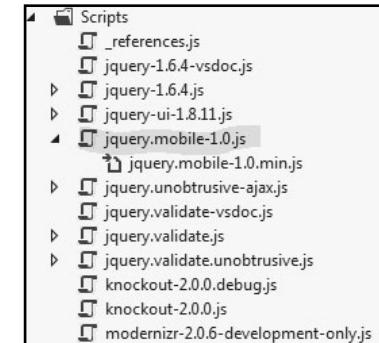


Рис. 10.15. Папка Scripts нового проекта

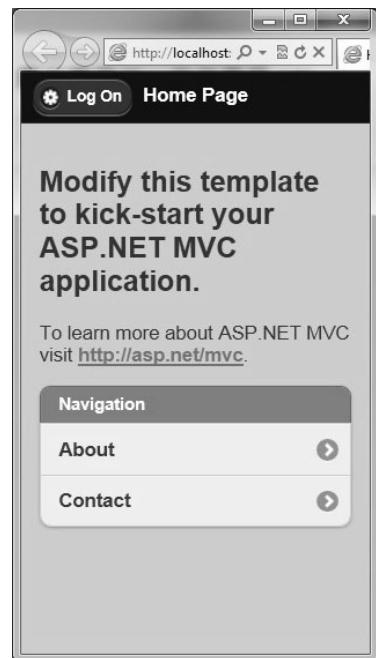


Рис. 10.13. Стандартная домашняя страница мобильного приложения

Как упоминалось ранее, значение "width-device-width" автоматически устанавливает ширину контента равной ширине экрана устройства:

```
<meta name="viewport" content="width=device-width" />
```

В качестве альтернативы для ширины окна просмотра может быть указано любое фиксированное значение в пикселях. Например, следующий код устанавливает начальную ширину страницы в 320 пикселей:

```
<meta name="viewport" content="width=320px" />
```

Приведенный ниже дескриптор включает в страницу стили jQuery Mobile. Он также позволяет конфигурировать темы через инфраструктуру управления темами jQuery (<http://jquerymobile.com/demos/1.0/docs/api/themes.html>):

```
<link rel="stylesheet"  
      href="@Url.Content("~/Content/jquery.mobile-1.0b2.min.css")" />
```

Наконец, данный дескриптор включает на страницу саму инфраструктуру jQuery Mobile Framework. Это открывает доступ при написании сценариев к выполнению операций AJAX, анимации, проверке достоверности и т.п.:

```
<script type="text/javascript"  
       src="@Url.Content("~/Scripts/jquery.mobile-1.0b2.min.js")" />  
</script>
```

Теперь давайте рассмотрим модифицированную HTML-разметку для страницы, которая включает несколько новых атрибутов. Инфраструктура jQuery Mobile идентифицирует разнообразные элементы, такие как страницы, кнопки, поля со списками и т.д., с помощью атрибутов `data-role`. Взглянув на дескриптор `body`, можно увидеть, что стандартный шаблон декорирован определенными дескрипторами `<div>` с атрибутами `data-role`:

```
<body>  
  <div data-role="page" data-theme="b">  
    <div data-role="header">  
      @if (IsSectionDefined("Header")) {  
        @RenderSection("Header")  
      } else {  
        <h1>@ViewBag.Title</h1>  
        @Html.Partial("_LogOnPartial")  
      }  
    </div>  
    <div data-role="content">  
      @RenderBody()  
    </div>  
  </div>  
</body>
```

Первый дескриптор `<div>` имеет атрибут `data-role="page"`, который идентифицирует `<div>` как одиночную страницу в мобильном приложении. Аналогично, заголовок страницы идентифицирован атрибутом `data-role="header"`, а тело – атрибутом `data-role="content"`.

В jQuery Mobile определены различные атрибуты для известных HTML-элементов, таких как `<H1>`, `<H2>`, `<P>` и `<table>`, а также списковых элементов и элементов форм наподобие кнопок, текстовых полей, списков выбора и т.д. За дополнительными сведениями обращайтесь на веб-сайт jQuery Mobile (<http://jQueryMobile.com>), на котором доступна подробная документация, демонстрационные приложения и многое другое.

Резюме

В этой главе рассказывалось о разнообразных аспектах программирования для мобильной веб-сети, в том числе то, что в действительности означает “мобильная веб-сеть”, и чем мобильные веб-сайты отличаются от настольных веб-сайтов. Рассматривались различные доступные инфраструктуры и технологии разработки, которые помогают сделать процесс построения мобильных веб-приложений более продуктивным. Было также показано, как предоставить наилучший пользовательский интерфейс за счет применения ряда доступных возможностей браузеров.

В главе затрагивались многие мобильные функциональные возможности ASP.NET MVC 4, включая перечисленные ниже.

- Внесение усовершенствований в стандартный шаблон Mobile Application.
- Возможность настройки стандартного шаблона за счет переопределения компоновки, представлений и частичных представлений.
- Специфичная для браузера поддержка (такая как представления, предназначенные специально для iPhone) и возможность переопределения функциональности браузера.
- Улучшение мобильного представления с использованием инфраструктуры jQuery Mobile Framework.

Выход за стандартные рамки

В этой части...

Глава 11. Параллельные, асинхронные и операции
над данными в реальном времени

Глава 12. Кеширование

Глава 13. Технологии оптимизации клиентской стороны

Глава 14. Расширенная маршрутизация

Глава 15. Многократно используемые компоненты
пользовательского интерфейса

Параллельные, асинхронные и операции над данными в реальном времени

Модель программирования веб-приложений традиционно была основана на синхронных коммуникациях типа “клиент-сервер”, при которых браузер выполняет HTTP-запрос и ожидает до тех пор, пока сервер не возвратит ответ. Хотя такая модель хорошо работает в большинстве сценариев, она может быть весьма неэффективной при поддержке длительно выполняющихся или сложных транзакций.

В этой главе будет показано, как задействовать мощные возможности асинхронной и параллельной обработки в ASP.NET MVC, чтобы иметь дело с более сложными сценариями, такими как обработка асинхронных запросов и применение коммуникаций реального времени для отправки и получения сообщений множеству одновременно подключенных клиентов.

Асинхронные контроллеры

Когда поступает запрос, ASP.NET извлекает один поток из пула для обработки этого запроса. Если процесс является синхронным, поток будет блокироваться от обработки других входящих запросов до тех пор, пока текущий процесс не будет завершен.

В большинстве сценариев выполняемый процесс является достаточно кратковременным, чтобы инфраструктура ASP.NET могла поддерживать несколько блокированных потоков. Тем не менее, если приложение должно обрабатывать большое количество входящих запросов или имеется очень много длительно выполняющихся запросов, пул потоков может исчерпаться, и возникнет условие, которое называется *нехваткой потоков*. Когда это происходит, веб-сервер начнет ставить в очередь новые входящие запросы. В какой-то момент очередь заполнится, и любые поступающие запросы будут отклоняться, возвращая код состояния HTTP 503 (сервер слишком занят).

Чтобы предотвратить полное израсходование пула потоков, контроллеры ASP.NET MVC могут быть настроены на асинхронное выполнение вместо синхронного, принятого по умолчанию. Использование асинхронного контроллера не изменяет количество времени, требуемого для запроса. Оно просто освобождает поток, выполняющий запрос, так что он может быть возвращен обратно в пул потоков ASP.NET.

Рассмотрим шаги, необходимые для обработки асинхронного запроса. Инфраструктура ASP.NET извлекает поток из пула и выполняет его для обработки входящего

запроса. После вызова действия ASP.NET MVC асинхронным образом поток возвращается в пул потоков, поэтому он может обрабатывать другие запросы. Асинхронная операция выполняется в другом потоке; когда она завершается, она уведомляет об этом ASP.NET. Инфраструктура ASP.NET снова извлекает поток из пула (это может быть другой поток, отличный от исходного) и вызывает его для завершения обработки запроса. Это включает также и процесс визуализации (вывода).

Создание асинхронного контроллера

Создать асинхронный контроллер довольно просто. Просто унаследуйте класс контроллера от базового класса `AsyncController`, который предоставляет методы для управления обработкой асинхронного запроса:

```
public class SearchController : AsyncController
{
}
```

Асинхронный контроллер необходим потому, что метод `SearchForBids()` использует сложный запрос LINQ, который может потребовать нескольких секунд на обработку:

```
public ActionResult SearchForBids(DateTime startingRange, DateTime endingRange)
{
    var bids = _repository
        .Query<Bid>(x => x.Timestamp >= startingRange
                      && x.Timestamp <= endingRange)
        .OrderByDescending(x => x.Timestamp)
        .ToArray();

    return Json(bids, JsonRequestBehavior.AllowGet);
}
```

До выхода версии ASP.NET MVC 4 при создании методов асинхронного контроллера должны были быть удовлетворены следующие соглашения.

- *ИмяДействияAsync*. Метод, возвращающий `void`; он начинает асинхронный процесс.
- *ИмяДействияCompleted*. Этот метод вызывается, когда асинхронный процесс завершен; он поддерживает возврат `ActionResult`.

Ниже приведен обновленный метод `SearchForBids()`, который был настроен на применение `BackgroundWorker` для асинхронного поиска предложений цены:

```
public void SearchForBidsAsync(DateTime startingRange, DateTime endingRange)
{
    AsyncManager.OutstandingOperations.Increment();

    var worker = new BackgroundWorker();
    worker.DoWork += (o, e) => SearchForBids(Id, e);
    worker.RunWorkerCompleted += (o, e) =>
    {
        AsyncManager.Parameters["bids"] = e.Result;
        AsyncManager.OutstandingOperations.Decrement();
    };
    worker.RunWorkerAsync();
}
```

```

private void SearchForBids(string Id, DoWorkEventArgs e)
{
    var bids = _repository
        .Query<Bid>(x => x.Timestamp >= startingRange
            && x.Timestamp <= endingRange)
        .OrderByDescending(x => x.Timestamp).ToList();

    e.Result = bids;
}

public ActionResult SearchForBidsCompleted(IEnumerable<Bid> bids)
{
    return Json(bids, JsonRequestBehavior.AllowGet);
}

```

Обратите внимание на вызов метода `Increment()` на свойстве `AsyncManager.OutstandingOperations` перед началом операции и вызов метода `Decrement()`, когда операция завершена. Это требуется для того, чтобы уведомить ASP.NET о том, сколько ожидающих операций содержит метод. Когда значение свойства `OutstandingOperations` достигает нуля, ASP.NET завершает асинхронную обработку метода и вызывает `SearchForBidsCompleted()`.

Как видите, кода немало. К счастью, в версии .NET Framework 4.5 появились новые ключевые слова `async` и `await`, которые помогают существенно упростить асинхронное программирование.



Дополнительные сведения об асинхронном программировании в .NET 4.5 доступны по следующему адресу: [http://msdn.microsoft.com/en-us/library/hh191443\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/hh191443(v=vs.110).aspx).

Ниже показана финальная версия метода `SearchForBids()`, модифицированная для использования упомянутых выше ключевых слов:

```

public async Task<ActionResult> SearchForBids(string Id)
{
    var bids = await Search(Id);
    return Json(bids, JsonRequestBehavior.AllowGet);
}

private async Task<IEnumerable<Bid>> Search(string Id)
{
    var bids = _repository
        .Query<Bid>(x => x.Timestamp >= startingRange
            && x.Timestamp <= endingRange)
        .OrderByDescending(x => x.Timestamp).ToList();
    return bids;
}

```

Действия контроллера, возвращающие экземпляр `Task`, могут быть сконфигурированы с тайм-аутом. Для установки тайм-аута применяется атрибут `AsyncTimeout`. В следующем примере действие контроллера имеет тайм-аут, равный 2500 миллисекунд. По истечении этого тайм-аута возвращается представление `AjaxTimedOut`:

```

[AsyncTimeout(2500)]
[HandleError(ExceptionType = typeof(TaskCanceledException), View = "AjaxTimedOut")]
public async Task<ActionResult> SearchForBids(string Id)
{
}

```

Обстоятельства, при которых используются асинхронные контроллеры

Никаких жестких правил, регламентирующих использование асинхронных действий, не существует. Приведенные ниже руководства помогут принять обоснованное решение о том, когда применять асинхронные действия.

Вот типичные сценарии, при которых лучше использовать синхронные действия:

- простые и кратко выполняющиеся операции;
- случаи, когда простота более важна, чем эффективность;
- интенсивные в плане работы с процессором операции (асинхронные операции в таких случаях не дают никаких преимуществ и могут даже добавить накладные расходы).

А вот типичные сценарии, при которых предпочтительнее применять асинхронные действия:

- длительно выполняющиеся операции, которые приводят к возникновению узкого места, связанного с производительностью;
- операции, интенсивные в плане работы с сетью или вводом-выводом;
- когда приложению требуется предоставить пользователям возможность отмены длительно выполняющихся операций.

Асинхронные коммуникации реального времени

Сеть World Wide Web является постоянно меняющейся средой; модели приложений, которые работали даже несколько месяцев назад, могут больше не удовлетворять ожиданиям пользователей. Вместо построения монолитных веб-приложений с десятками страниц все большее число разработчиков предпочитают создавать приложения с применением одностраничной архитектуры или небольшого набора страниц, которые динамически обновляются в реальном времени.

Адаптация технологий работы с данными в реальном времени может быть объяснена взрывоподобным распространением социальных сетей и мобильных устройств. В современном мире люди постоянно находятся в движении, и им нужен мгновенный доступ к последней информации, будь то спортивные результаты любимой команды, цены на акции или новые сообщения от друзей. И поскольку все большее количество людей посещают веб-сайты из мобильных устройств, очень важно, чтобы веб-приложение было в состоянии определять доступность сети и элегантно поддерживать функциональные возможности веб-браузеров на множестве разных устройств.

Сравнение моделей приложений

Модель традиционного веб-приложения опирается на синхронные коммуникации. По мере того как пользователь взаимодействует с приложением, веб-браузер выполняет запросы к серверу, который обрабатывает их и возвращает снимок текущего состояния приложения. Так как нет никакой гарантии, что пользователь запустит еще один запрос, существует высокая вероятность того, что просматриваемый им контент может устареть, приводя к конфликту данных.

Использование технологий, подобных AJAX, решает только часть проблемы. В большинстве случаев пользователь по-прежнему должен инициировать запрос. Технология AJAX полагается на традиционный подход “запрос/ответ”, при котором

взаимодействия являются транзакционными и атомарными, а со всем тем, что меняется за пределами текущей транзакции, взаимодействовать невозможно — чтобы вернуть синхронизацию, потребуется выполнить еще один запрос. Таким образом, этот подход не позволяет хорошо обрабатывать обновления в реальном времени. Для поддержки подобных сценариев должны применяться более развитые приемы, которые создают долговременные взаимодействия между сервером и браузером.

Давайте рассмотрим различные доступные коммуникации реального времени. Имейте в виду, что протокол HTTP спроектирован на основе шаблона коммуникаций “запрос/ответ” и не поддерживает непосредственно возможность сервера взаимодействовать с клиентом без предварительной отправки запроса со стороны клиента.

Опрос HTTP

На рис. 11.1 показано создание непрерывного взаимодействия за счет имитации “постоянного подключения” к серверу на базе последовательности стандартных запросов AJAX. Это обычно достигается отправкой запросов AJAX на регулярной основе с использованием таймера JavaScript.

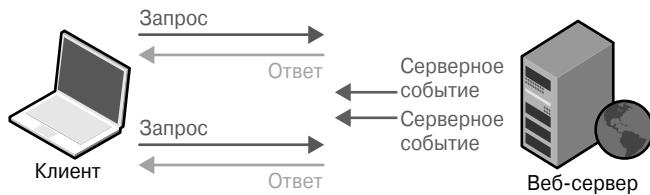


Рис. 11.1. Опрос HTTP

На рис. 11.1 можно видеть опрос в действии. Наиболее важный аспект, касающийся этой технологии, заключается в том, что браузер создает новый запрос немедленно после завершения каждого существующего запроса (независимо от того, успешно ли завершился запрос и содержит ли он данные), т.е. поддерживается встроенная отказоустойчивость.

Таким образом, опрос является одним из наиболее надежных и безопасных в отношении сбоев методов коммуникаций “реального времени”, однако за эту надежность приходится платить. Как показано на рис. 11.2, опрос генерирует относительно большой объем сетевого трафика и загрузки сервера, особенно учитывая то, что запросы обрабатываются независимо от наличия на сервере обновлений (поэтому множество, если не подавляющее большинство, запросов не будут возвращать данные).

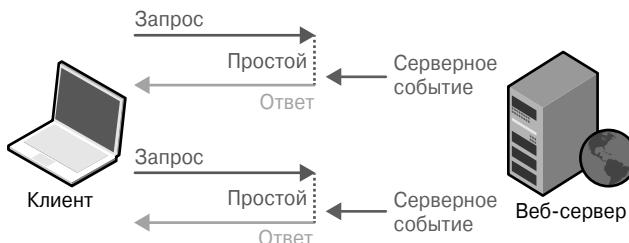


Рис. 11.2. Длительный опрос HTTP

Поддержка браузеров

Опрос использует разнообразные браузерные технологии, существующие с момента появления графического веб-браузера, поэтому он работает везде и всегда при условии, что поддержка JavaScript включена.

Недостатки

Опрос обладает рядом недостатков. Чрезмерное количество запросов по сравнению с объемом передаваемых действительных данных делает эту технологию чрезвычайно расточительной. Клиентские запросы и серверные события не всегда синхронизированы; существует возможность возникновения множества серверных событий между клиентскими запросами. Если ослабить контроль, этот подход может неумышленно создать на ваших серверах условия, похожие на атаку типа отказа в обслуживании!

Длительный опрос HTTP

Технология длительного опроса HTTP является главным образом реализацией на стороне сервера, при которой браузер отправляет запрос AJAX серверу с целью извлечения данных, а сервер сохраняет подключение открытым до тех пор, пока не появятся данные для возврата. Это радикально отличается от традиционного подхода “запрос/ответ”, когда сервер немедленно отвечает, что у него нет данных, если он не в состоянии предоставить данные при получении запроса AJAX.

Длительный опрос влечет за собой выполнение запроса в ожидании возможного будущего серверного события. Вместо немедленного возврата сервером ответа входящий запрос блокируется до тех пор, пока не произойдет серверное событие либо не истечет время тайм-аута запроса (или не разорвется подключение). Затем клиент должен инициировать новый запрос длительного опроса, чтобы начать следующее взаимодействие и продолжить извлечение обновленных данных.

Поддержка браузеров

Из-за наличия множества разных реализаций длительного опроса эта технология работает (с разной степенью надежности) во всех браузерах.

Недостатки

Поскольку инфраструктура Интернета построена вокруг простых взаимодействий через запросы и ответы HTTP и не предназначена для обработки долго существующих подключений, запросы длительного опроса не являются надежными, т.к. они имеют тенденцию часто отключаться. Разорванные подключения на самом деле являются частью рабочего потока длительного опроса, поэтому они обрабатываются точно так же, как успешный запрос: начинается новый запрос. Однако эта сложность увеличивает ненадежность подхода в целом.

Кроме того, как и большинство широкоподдерживаемых технологий, реализации длительного опроса часто ограничены наименьшим общим набором функциональных возможностей, поддерживаемым всеми браузерами, который сводится к простому HTTP-запросу GET (URL, который может быть применен к дескриптору IFRAME или <script>).

События, отправляемые сервером

Подход с событиями, отправляемыми сервером (известный также под названием “EventSource”), очень похож на длительный опрос тем, что клиент передает запрос

HTTP серверу, а результирующее подключение остается открытым до тех пор, пока сервер не будет иметь данные, удовлетворяющие клиентскому запросу. Фундаментальное отличие между этими двумя подходами заключается в том, что подход с событиями, отправляемыми сервером, не закрывает подключение после возврата начального ответа сервера. Вместо этого сервер сохраняет подключение открытым с целью отправки клиенту дополнительных обновлений, когда они станут доступными (рис. 11.3).

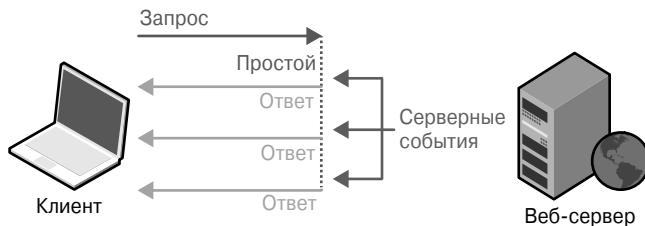


Рис. 11.3. События, отправляемые сервером

Обратите внимание, что подход с событиями, отправляемыми сервером, упрощает однонаправленные коммуникации от сервера к клиенту. То есть клиент не имеет возможности отправить дополнительную информацию обратно серверу после начального запроса в том же самом подключении. Чтобы клиент мог взаимодействовать с сервером, он должен выполнять дополнительные запросы AJAX. Тем не менее, клиент не обязан закрывать канал событий, отправляемых сервером, чтобы делать эти дополнительные запросы – клиент может использовать стандартные технологии AJAX для отправки информации серверу, а сервер может отвечать на эти события через открытый канал событий, отправляемых сервером, либо другой запрос AJAX (или обоими способами).

Часть “EventSource” относится к API-интерфейсу JavaScript под названием EventSource, который представляет собой API-интерфейс клиентской стороны (определенный в виде части более крупной спецификации HTML 5), упрощающий реализацию в браузере подхода с событиями, отправляемыми сервером.

Поддержка браузеров

Большинство основных браузеров имеют какую-то разновидность поддержки событий, отправляемых сервером, с заметным исключением в форме Internet Explorer. В частности, встроенная поддержка доступна в Chrome 9+, Firefox 6+, Opera 11+ и Safari 5+.

Недостатки

Хотя этот подход позволяет серверу доставлять обновления реального времени, он допускает только однонаправленные коммуникации от клиента к серверу. Это значит, что открытый канал не является двунаправленным. Тем не менее, клиенты по-прежнему могут взаимодействовать с сервером, выдавая дополнительные запросы AJAX.

Веб-сокеты

API-интерфейс веб-сокетов (WebSocket API) – это новый протокол (предложенный как часть спецификации HTML 5), который эффективно преобразует стандартные подключения запросов HTTP в двунаправленные, дуплексные коммуникационные каналы TCP. Последние (не так широко распространенные) версии этого протокола также предлагают опцию защищенных коммуникаций. Работа веб-сокетов проиллюстрирована на рис. 11.4.

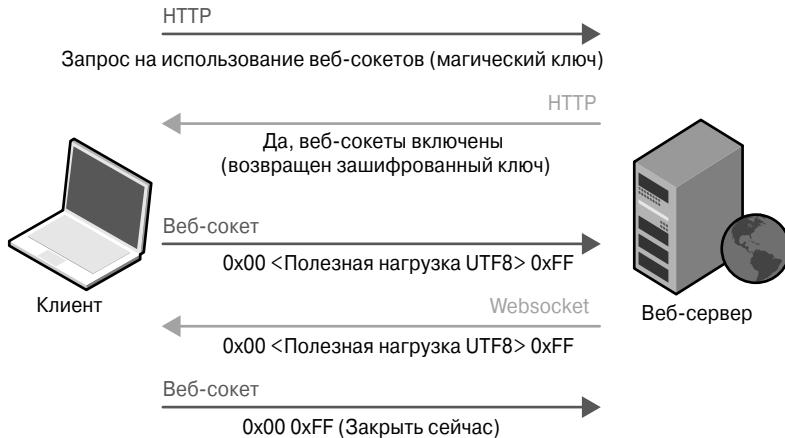


Рис. 11.4. Веб-сокеты

Поддержка браузеров

Большинство основных браузеров имеют какую-то разновидность поддержки веб-сокетов, с заметными исключениями в форме Internet Explorer и Opera. В частности, встроенная поддержка доступна в IE 10, Firefox 6.0+, Chrome 4.0+, Safari 5.0+ и iOS 4.2+.



В случае, когда браузеры не имеют собственной реализации веб-сокетов, могут применяться определенные тонкие прослойки (такие как `web-socket-js`, которая использует реализацию на Flash). Эти тонкие прослойки едва ли можно считать “собственными”, однако они помогают заполнить пробелы.

Недостатки

Хотя адаптация поддержки веб-сокетов в браузерах растет, она пока еще не реализована во всех основных браузерах. Кроме того, просто наличие поддержки веб-сокетов не гарантирует ее работу для каждого пользователя. Антивирусные программы, брандмауэры и HTTP-прокси могут помешать подключениям веб-сокетов, иногда даже их вообще бесполезными.

Расширение возможностей коммуникаций реального времени

Добавление коммуникаций реального времени к веб-приложению может оказаться нетривиальной задачей для разработчиков, занимающихся этим в одиночку. К счастью, в Microsoft осознали необходимость в коммуникациях реального времени и отреагировали включением в состав версии .NET Framework 4.0 библиотек для параллельной обработки на уровне задач, а также созданием *SignalR* – библиотеки асинхронных уведомлений с открытым кодом для ASP.NET.

Библиотека SignalR упрощает встраивание коммуникаций реального времени в веб-приложение. Она действует в качестве абстракции поверх подключения HTTP и предлагает разработчикам на выбор две модели программирования: концентраторы и постоянные подключения. Эта библиотека состоит из серверного API-интерфейса и клиентских библиотек для .NET и JavaScript.

Библиотека SignalR поддерживает несколько разных транспортных моделей. Каждая модель решает, каким образом отправлять и получать данные, и как клиент и сервер подключаются и отключаются. По умолчанию SignalR будет выбирать “лучшую” транспортную модель на основе того, что поддерживается размещающим браузером (разработчики также имеют возможность выбора специфичного транспорта).

Библиотека SignalR поддерживает следующие транспортные модели:

- веб-сокеты;
- события, отправляемые сервером;
- постоянные фреймы;
- длительный опрос.

Проще всего приступить к работе с библиотекой SignalR, воспользовавшись NuGet для установки ее пакета:

```
Install-Package SignalR
```

Постоянные подключения

После установки пакета можно начинать построение приложения с коммуникациями реального времени. Давайте сначала посмотрим, как настраивать постоянное подключение для организации отправки сообщений между клиентом и сервером.

Первым делом необходимо создать специальный объект подключения, унаследовав его от базового класса `PersistentConnection`. Ниже приведен пример специального класса подключения. Обратите внимание на переопределенный метод `OnReceivedAsync()`, который отправляет широковещательное сообщение всем клиентам, подключенным к серверу:

```
using System.Threading.Tasks;
using SignalR;

public class EbuyCustomConnection : PersistentConnection
{
    protected override Task OnReceivedAsync(IRequest request,
                                            string connectionId, string data)
    {
        // Отправить широковещательное сообщение всем клиентам.
        return Connection.Broadcast(data);
    }
}
```

Следующий шаг заключается в регистрации специального подключения, добавив его в таблицу маршрутов ASP.NET MVC (удостоверьтесь, что во избежание конфликтов отображение SignalR находится перед всеми остальными маршрутами).

```
RouteTable.Routes.MapConnection<EbuyCustomConnection>("echo", "echo/{*operation}");
```

Теперь необходимо добавить на стороне клиента ссылку на обязательные JavaScript-файлы SignalR:

```
<script src="http://code.jquery.com/jquery-1.7.js"
       type="text/javascript"></script>
<script src="Scripts/jquery.signalR-0.5.0.min.js"
       type="text/javascript"></script>
<script type="text/javascript">
```

Чтобы получить сообщение, инициализируйте объект подключения и подпишите его на событие `received`. Финальный шаг состоит в запуске подключения за счет вызова `connection.start()`:

```
$ (function () {
    var connection = $.connection('/echo');
    connection.received(function (data) {
        $('#messages').append('<li>' + data + '</li>');
    });
    connection.start();
});
```

Для отправки сообщения вызовите метод `send()` на объекте подключения:

```
connection.send("Hello SignalR!");
```

Концентраторы

Использовать концентраторы намного проще, чем создавать специальные низкоуровневые объекты подключений. Концентраторы предоставляют инфраструктуру удаленного вызова процедур (Remote Procedure Call – RPC), построенную поверх класса `PersistentConnection`. Концентраторы должны применяться вместо специальных объектов подключений, чтобы избегать необходимости в прямой обработке сообщений диспетчеризации.

В отличие от объектов `PersistentConnection`, концентраторы не требуют какого-то специфичного конфигурирования маршрутизации, поскольку они доступны посредством специального URL (`/signalr`).

Создание специального концентратора – процесс простой: нужно лишь построить класс, унаследованный от базового класса `Hub`, и добавить метод для отправки сообщений. Затем с помощью динамически типизированного экземпляра `Clients`, доступного в базовом классе, можно определить специальный метод (например, `DisplayMessage`), который будет использоваться для взаимодействия с клиентами, подключенными к концентратору:

```
public class EbuyCustomHub : Hub
{
    public void SendMessage(string message)
    {
        Clients.displayMessage(message);
    }
}
```

Чтобы взаимодействовать с концентратором, сначала понадобится добавить необходимые JavaScript-файлы SignalR:

```
<script src="Scripts/jquery-1.6.2.min.js"
       type="text/javascript"></script>
<script src="Scripts/jquery.signalR-0.5.0.min.js"
       type="text/javascript"></script>
<script src="/signalr/hubs"
       type="text/javascript"></script>
```

Для получения сообщения создайте экземпляр прокси-класса JavaScript концентратора и подпишите его на один или более методов, определенных динамическим объектом `Clients`. Вызовите `$.connection.hub.start()` для инициализации коммуникационного канала между клиентом и сервером:

```
$(function () {
    // Прокси, создаваемый на лету.
    var proxy = $.connection.ebuyCustomHub;
    // Объявить функцию обратного вызова.
    proxy.displayMessage = function (message) {
        $('#messages').append('<li>' + message + '</li>');
    };
    // Запустить подключение.
    $.connection.hub.start();
});
```

Чтобы отправить сообщение, вызовите один из открытых методов, определенных для концентратора (например, `sendMessage()`):

```
connection.sendMessage("Hello SignalR!");
```

Концентраторы исключительно расширяемы. Довольно легко создать множество разных методов для обработки различных типов серверных событий. Кроме того, в дополнение к отправке строковых сообщений, между клиентом и сервером допускается передача объектов JSON.

Ниже приведен пример концентратора, который поддерживает несколько типов сообщений:

```
public class EbuyCustomHub : Hub
{
    public void PlaceNewBid(string jsonObject)
    {
        var serializer = new JavaScriptSerializer();
        var bid = serializer.Deserialize<Bid>(jsonObject);
        Clients.newBidPosted(bid);
    }

    public void AuctionClosed(Auction auction)
    {
        Clients.auctionClosed(auction);
    }
}
```



Существующие типы .NET, отправленные объекту `Clients`, будут автоматически сериализоваться в формат JSON. Входящие сообщения требуют ручной десериализации с применением сериализатора JSON.

В следующем фрагменте приведен код JavaScript, используемый образцовым приложением Ебуу для получения и отправки уведомлений о новых предложениях цены. Чтобы отправить новое предложение цены, необходимо создать объект JSON и сериализовать его в строку; входной параметр события концентратора (`bid`) будет передаваться как объект JSON:

```
$(function () {
    // Прокси, создаваемый на лету.
    var proxy = $.connection.ebuyCustomHub;
    // Объявить серверные методы обратного вызова.
    proxy.newBidPosted = function (bid) {
        $('#bids').append('<li>Auction: ' + bid.Auction.Title + ' Latest Bid: '
            + bid.Amount.Value + ' </li>');
    };
});
```

```

// Запустить подключение.
$.connection.hub.start();

$("#postBid").click(function () {
    var bidToPost = GetBid();
    proxy.placeNewBid(bidToPost);
});

function GetBid() {
    var bidPrice = $('#bidPrice').val();
    var newBid = "{ 'Code': 'USD', 'Value': '" + bidPrice + "' }";
    return "{ 'Auction': { 'Id': '61fdb6eb-b565-4a63-b048-0418dcb8b28d',
        'Title': 'XBOX 360' }, 'Amount': " + newBid + " }";
}

```

Конфигурирование и настройка

В отличие от традиционных веб-приложений, которые требуют кратко действующих подключений, для коммуникаций реального времени необходимы подключения, существующие значительно дольше. Для эффективной настройки библиотеки SignalR понадобятся соответствующий мониторинг и конфигурирование, которые позволят получить лучшую производительность и сбалансировать требования этой библиотеки к ресурсам относительно остальных нужд веб-приложения.

Управление подключениями SignalR

Исполняющая среда SignalR предоставляет интерфейс `IConfigurationManager`, который может использоваться при настройке параметров подключения для SignalR. Конфигурируемые параметры подключения перечислены в табл. 11.1.

Таблица 11.1. Конфигурируемые параметры SignalR

Параметр	Описание
ConnectionTimeout	Промежуток времени, в течение которого простоявавшее подключение удерживается открытым, прежде чем будет закрыто (по умолчанию составляет 110 секунд)
DisconnectTimeout	Промежуток времени после закрытия подключения, в течение которого производится ожидание перед выдачей клиенту события разрыва (по умолчанию составляет 20 секунд)
HeartBeatInterval	Интервал проверки состояния подключения (по умолчанию составляет 10 секунд)
KeepAlive	Промежуток времени, в течение которого производится ожидание перед отправкой ping-сигнала для простоявшего подключения (по умолчанию составляет 30 секунд). Если подключение является активным, ConnectionTimeout не дает никакого эффекта; для отключения потребуется установить в null

Указание параметров SignalR должно осуществляться в течение начального запуска веб-приложения, например:

```

// Изменить тайм-аут подключения на 60 секунд.
GlobalHost.Configuration.ConnectionTimeout = TimeSpan.FromSeconds(60);

```

Конфигурирование среды

По умолчанию ASP.NET и Internet Information Services (IIS) сконфигурированы для обеспечения наилучшей масштабируемости при управлении множеством запросов в секунду. Для поддержки коммуникаций реального времени понадобится модифицировать несколько параметров, чтобы корректно обрабатывать большое количество параллельных подключений.

Для увеличения максимального числа параллельных запросов, обрабатываемых IIS, откройте окно командной строки с привилегиями администратора и перейдите в каталог %windir%\System32\inetsrv\. Затем запустите следующую команду, чтобы изменить принятое в IIS 7 стандартное значение appConcurrentRequestLimit, равное 5 000 подключений, на 100 000 подключений:

```
appcmd.exe set config /section:serverRuntime /appConcurrentRequestLimit:100000
```

По умолчанию ASP.NET 4.0 конфигурируется для поддержки 5 000 подключений на процессор. Чтобы поддерживать дополнительные подключения на процессор, измените значение параметра maxConcurrentRequestsPerCPU в файле aspnet.config:

```
<system.web>
  <applicationPool maxConcurrentRequestsPerCPU="20000" />
</system.web>
```

Этот файл находится в системном каталоге .NET Framework (%windir%\Microsoft .NET\Framework\v4.0.30319 для 32-разрядной и %windir%\Microsoft .NET\Framework64\v4.0.30319 для 64-разрядной операционной системы).

Исполняющая среда ASP.NET начнет регулировать запросы, используя очередь, когда общее количество подключений превысит максимальное число параллельных запросов на процессор (т.е. maxConcurrentRequestsPerCPU – количество логических процессоров на машине). Для управления регулирующей очередью необходимо изменять значение параметра requestQueueLimit в файле machine.config (расположенном в том же месте, где и aspnet.config).

Чтобы модифицировать предельный размер очереди запросов, установите атрибут autoConfig элемента processModel в false и обновите размер requestQueueLimit:

```
<processModel autoConfig="false" requestQueueLimit="250000" />
```

Резюме

В этой главе было показано, как внедрять распараллеливание при проектировании и построении веб-приложения. В ней также рассказывалось о том, как использовать асинхронные контроллеры для обработки длительно выполняющихся запросов, и каким образом применять библиотеку SignalR для реализации внутри веб-приложения коммуникаций реального времени.

Кеширование

Практически каждый веб-сайт обслуживает определенный объем контента, который изменяется нечасто, будь то статические страницы, модифицируемые только при обновлении приложения, или страницы контента, изменяемые раз в несколько дней или даже часов.

Проблема заключается в том, что веб-приложение выполняет интенсивную работу по генерации такого контента с нуля каждый раз, когда он запрашивается, оставаясь в неведении относительно того, что одни и те же страницы генерировались, возможно, сотни раз. Не будет ли более разумным попытаться избежать многократной генерации одного и того же контента, обеспечив вместо этого однократную генерацию и сохранение контента и затем его повторное использование в последующих запросах?

Концепция сохранения и многократного использования сгенерированных данных называется *кешированием* и представляет собой один из наиболее эффективных путей улучшения производительности веб-приложения. Однако то, какой контент должен быть кеширован, и насколько долго он хранится в кеше – это обычно не является предметом, который веб-приложение может выяснить самостоятельно. Вместо этого вы должны предоставить приложению информацию, необходимую для того, чтобы определить контент, являющийся кандидатом на кеширование.

К счастью, как ASP.NET Framework, так и ASP.NET MVC Framework, предлагают несколько API-интерфейсов кеширования для удовлетворения всех потребностей в этой функциональности. В настоящей главе описаны различные технологии кеширования и доступные API-интерфейсы, а также способы применения этих приемов для улучшения производительности приложений ASP.NET MVC.

Типы кеширования

В целом технологии кеширования веб-приложений делятся на две категории: кеширование серверной стороны и кеширование клиентской стороны. Хотя обе категории преследуют одну и ту же цель – ограничение объема дублированного контента, который должен генерироваться и передаваться по сети, основное отличие между ними касается того, где сохраняются кешированные данные – на сервере или в браузере клиента.

Кеширование серверной стороны

Технологии кеширования серверной стороны сосредоточены на оптимизации способа, по которому сервер извлекает, генерирует или каким-то другим образом манипулирует контентом. Главная цель кеширования серверной стороны заключается в

ограничении объема работы, связанной с обслуживанием запроса, то ли избегая вызовов для извлечения данных из базы, то ли даже сокращая количество процессорных циклов, которые необходимы для генерации HTML-разметки — учитывается каждый мельчайший аспект.

Ограничение объема работы по обслуживанию запроса не только уменьшает время, требуемое на завершение запроса, но также обеспечивает наличие большего количества серверных ресурсов, доступных для обработки большего числа запросов за то же самое время.

Кеширование клиентской стороны

В дополнение к кешированию контента на сервере, современные браузеры предлагают ряд собственных механизмов кеширования. Технологии клиентской стороны открывают совершенно новые возможности для улучшения производительности приложений, начиная с интеллектуального устранения дублированных запросов и заканчивая сохранением контента непосредственно в локальной среде пользователя.

В то время как главная цель кеширования серверной стороны связана с максимально быстрой и эффективной обработкой запросов, основная цель технологий кеширования клиентской стороны состоит в устраниении любых запросов вообще. Устранение ненужных запросов улучшает работу не только пользователей, которые их инициировали, но также помогает уменьшить общую нагрузку на сервер, улучшая работу всех пользователей, которые параллельно находятся на сайте.

Важно иметь в виду, что технологии кеширования серверной и клиентской стороны занимают свои ниши, и ни одна из них не является более важной, чем другая. Самые эффективные стратегии кеширования обычно предусматривают комбинацию этих двух типов технологий кеширования, чтобы попытаться извлечь лучшее из обоих.

Технологии кеширования серверной стороны

Когда дело доходит до кеширования серверной стороны, на выбор имеется множество технологий, начиная с простого хранилища внутри памяти и заканчивая выделенными кеширующими серверами. Фактически большинство вариантов кеширования, доступных в приложениях ASP.NET MVC, не относятся к ASP.NET MVC Framework, а принадлежат ядру ASP.NET Framework.

В последующих разделах рассматриваются технологии кеширования серверной стороны, которые наиболее часто применяются в контексте веб-приложений ASP.NET MVC.

Кеширование на уровне запросов

Каждый запрос ASP.NET начинается с того, что ASP.NET Framework создает новый экземпляр объекта `System.Web.HttpContext`, который действует в качестве центральной точки взаимодействия между компонентами на протяжении запроса.

Одним из множества свойств `HttpContext` является `HttpContext.Items` — словарь, который существует в течение времени жизни запроса и которым может манипулировать любой компонент.

Доступность этого свойства — вместе с тем фактом, что его область действия всегда ограничена текущим запросом — делает словарь `Items` великолепным местом для хранения данных, имеющих отношение к текущему запросу. Его доступность также предоставляет компонентам хороший способ передачи данных друг другу в слабо свя-

занной манере, используя общий объект `HttpContext.Items` в качестве посредника вместо того, чтобы взаимодействовать напрямую.

Поскольку это просто интерфейс `IDictionary`, работа с коллекцией `Items` предельно проста. Например, вот как сохранять данные в этой коллекции:

```
HttpContext.Items["IsFirstTimeUser"] = true;
```

Извлечение данных из словаря тоже осуществляется легко:

```
bool isFirstTimeUser = (bool)HttpContext.Items["IsFirstTimeUser"];
```

Обратите внимание, что словарь `Items` не является строго типизированным, поэтому перед использованием хранимый объект сначала понадобится привести к желаемому типу.

Кеширование на уровне пользователей

Состояние сеанса ASP.NET позволяет запоминать данные, которые сохраняются между несколькими запросами. Состояние сеанса ASP.NET – это хранилище данных, которое приложения могут применять для сохранения информации на основе отдельных пользователей.

Когда сеансы включены, компоненты могут обращаться к свойству `HttpContext.Session` или `Session` и сохранять информацию для будущего запроса того же самого пользователя, а также извлекать информацию, которая была сохранена в предыдущих запросах этого пользователя.



Поскольку область действия состояния сеанса ASP.NET ограничивается текущим пользователем, состояние сеанса не может применяться для разделения информации между пользователями.

Подобно словарю `HttpContext.Item`, объект `Session` является нетипизированым словарем, поэтому взаимодействуют с ним аналогичным образом.

Например, в следующем коде показано, как сохранить внутри сеанса имя пользователя:

```
HttpContext.Session["username"] = "Hrusi";
```

А вот как выполнить извлечение и приведение нетипизированного значения:

```
string name = (string)HttpContext.Session["username"];
```

Время жизни сеанса

Объекты, сохраненные в `Session` пользователя, существуют до тех пор, пока сеанс не будет уничтожен сервером, обычно после того, как пользователь был неактивным в течение определенного периода времени.

Стандартную продолжительность тайм-аута, равную 20 минутам, можно легко изменить, модифицировав значение атрибута `timeout` параметра `sessionState` в скрипторе `system.web` внутри файла `web.config` приложения.

Например, в следующей конфигурации стандартный тайм-аут увеличивается с 20 до 30 минут:

```
<system.web>
  <sessionState timeout="30" />
</system.web>
```

Сохранение данных сеанса

В плане места хранения данных сеанса имеется довольно много гибкости. Стандартное поведение предполагает сохранение всей информации состояния сеанса в памяти, но для хранения данных можно также выбрать службу состояния сеанса ASP.NET (ASP.NET Session State Service), базу данных SQL Server или любой другой источник данных, реализовав для этого собственный поставщик.

Кэширование на уровне приложений

Инфраструктура ASP.NET предлагает класс `HttpApplicationState`, предназначенный для сохранения данных уровня приложения, который доступен через свойство `HttpContext.Application`. Свойство `HttpContext.Application` – это коллекция пар “ключ/значение”, похожая на `HttpContext.Items` и `HttpContext.Session`, но существующая на уровне приложения, так что данные, которые добавляются к ней, могут охватывать пользователей, сеансы и запросы.

Сохранение данных в `HttpApplicationState` осуществляется следующим образом:

```
Application["Message"] = "Welcome to EBuy!";
Application["StartTime"] = DateTime.Now;
```

А чтение данных, хранящихся в `HttpApplicationState`, производится так:

```
DateTime appStartTime = (DateTime)Application["StartTime"];
```

Данные, хранящиеся в `HttpApplicationState`, существуют на протяжении времени жизни рабочего процесса Internet Information Services, в котором размещен экземпляр приложения. Поскольку временем жизни рабочих процессов управляет IIS, а не ASP.NET, имейте в виду, что `HttpApplicationState` может оказаться ненадежным путем для сохранения и извлечения постоянных значений.

По этой причине коллекция `HttpApplicationState` должна использоваться, только когда данные гарантированно являются одинаковыми между всеми рабочими процессами. Например, если вы читаете контент из дискового файла или извлекаете из базы данных значения, которые редко изменяются, можете применять `HttpApplicationState` в качестве кеширующего уровня, тем самым устранив доро-гостоящие вызовы для получения упомянутых значений.

Кеш ASP.NET

Более удачной альтернативой сохранению данных уровня приложения в `HttpApplicationState` является использование объекта `System.Web.Cache`, доступного через свойство `HttpContext.Cache`.

`System.Web.Cache` – это хранилище на основе пар “ключ/значение”, которое действует очень похоже на `HttpContext.Items` и `HttpSessionState`; тем не менее, хранящиеся в нем данные не ограничены отдельными запросами или пользовательскими сессиями.

На самом деле объект `Cache` намного больше похож на `HttpApplicationState`, однако он способен пересекать границы рабочего процесса, таким образом устранивая большую часть сложностей, присущих `HttpApplicationState`, что и делает его в общем случае наилучшим вариантом.

Исполняющая среда ASP.NET автоматически управляет удалением кэшированных элементов и уведомляет приложение о том, что такое удаление произошло, поэтому

можно повторно выполнить наполнение данными. ASP.NET удаляет кешированные элементы в одной из следующих ситуаций:

- истек срок хранения кешированного элемента;
- изменились зависимости кешированного элемента, что привело к его недействительности;
- на сервере исчерпались ресурсы, поэтому он должен затребовать обратно память.

Истечение срока хранения

При добавлении элементов в Cache можно указывать, насколько долго будут существовать данные, прежде чем истечет срок их хранения и они больше не должны использоваться. Этот временной промежуток может быть выражен одним из двух способов.

- **Скользящее истечение.** Указывает, что срок хранения элемента должен истечь через определенное время после последнего доступа к нему. Например, если вы кешируете элемент со скользящим истечением, равным 20 минут, и приложение непрерывно обращается к элементу каждые несколько минут, то элемент должен оставаться в кеше неопределенно долго (предполагая, что кешированный элемент не имеет никаких зависимостей, а сервер не испытывает нехватки памяти). Как только приложение прекратит обращаться к элементу на протяжении, по крайней мере, 20 минут, срок хранения элемента истечет.
- **Абсолютное истечение.** Указывает, что срок хранения элемента истекает в определенный момент времени, независимо от того, насколько часто к нему производится обращение. Например, если вы кешируете элемент с абсолютным истечением 22:20:00, то элемент больше не будет доступен после 22:20:01.



Для каждого элемента может быть указано истечение только одного типа – скользящее или абсолютное. На одном и том же элементе нельзя использовать оба типа истечения.

Тем не менее, разные типы истечения можно применять для разных кешированных элементов.

Зависимости кеша

Время жизни элемента в кеше можно также сконфигурировать как зависимое от других элементов приложения, таких как файлы или базы данных. Когда элемент, от которого зависит элемент кеша, изменяется, ASP.NET удаляет этот элемент из кеша.

Например, если веб-сайт отображает отчет, который приложение создает из XML-файла, результирующий отчет можно поместить в кеш и сконфигурировать для него зависимость от этого XML-файла. Когда XML-файл изменяется, ASP.NET удаляет отчет из кеша. При следующем запросе этого отчета код сначала выясняет, находится ли отчет в кеше, и если нет, то повторно создает его. В результате гарантируется постоянная доступность актуальной версии отчета.

Зависимость от файла – не единственный тип зависимости, доступный в ASP.NET; здесь в готовом виде предлагаются все типы зависимостей, перечисленные в табл. 12.1, а также предоставляется возможность создавать собственные политики зависимостей.

Таблица 12.1. Политики зависимостей кеша

Тип зависимости	Описание
Объединенная	Объединяет несколько зависимостей (через класс <code>System.Web.Caching.AggregateCacheDependency</code>). Кэшированный элемент удаляется, когда изменяется любая зависимость внутри этого объединения
Специальная	Кэшированный элемент зависит от специального класса, производного от <code>System.Web.Caching.CacheDependency</code> . Например, можно создать специальную зависимость кеша от веб-службы, которая приводит к удалению данных из кеша, когда обращение к веб-службе дает в результате определенное значение
Файловая	Кэшированный элемент зависит от внешнего файла и удаляется из кеша, если файл модифицируется или уничтожается
Ключевая	Кэшированный элемент зависит от другого элемента в кеше приложения (ссылаемого посредством его ключа в кеше). Кэшированный элемент удаляется в случае удаления из кеша этого целевого элемента
SQL	Кэшированный элемент зависит от изменений в какой-то таблице базы данных Microsoft SQL Server. Кэшированный элемент удаляется при обновлении этой таблицы

Очистка

Очистка – это процесс удаления элементов из кеша, когда памяти оказывается недостаточно. Обычно удаляются элементы, к которым не было обращения в течение определенного времени или которые при добавлении в кеш были помечены как имеющие низкий приоритет. Для выяснения того, какие элементы очищать первыми, исполняющая среда ASP.NET использует объект `CacheItemPriority`.

Во всех случаях ASP.NET предоставляет делегат `CacheItemRemovedCallback` для уведомления приложения о том, что элемент находится в процессе удаления.

Кеш вывода

Хотя все упомянутые выше технологии кеширования были сосредоточены на кешировании данных, ASP.NET позволяет оперировать на более высоком уровне, кешируя HTML-контент, который генерируется как результат запроса. Такая технология называется *кешированием вывода*, и она является мощным средством, доступным еще со времен первой версии ASP.NET Framework.

Чтобы максимально возможно упростить кеширование вывода, ASP.NET MVC предоставляет `OutputCacheAttribute` – фильтр действий, который сообщает ASP.NET MVC о необходимости добавления в кеш вывода визуализированных результатов действия контроллера.

Участие действий контроллера в кешировании вывода сводится к снабжению нужного действия атрибутом `OutputCacheAttribute`. По умолчанию этот атрибут будет кешировать визуализированный HTML-контент со сроком хранения 60 секунд. При следующем запросе к действию контроллера после истечения срока хранения кешированного контента ASP.NET MVC выполнит действие повторно и кеширует сгенерированный им HTML-контент еще раз.

Чтобы посмотреть на работу кеширования вывода ASP.NET MVC, попробуем добавить `OutputCacheAttribute` к действию контроллера в образцовом приложении `EBuy`:

```
[OutputCache(Duration=60, VaryByParam="none")]
public ActionResult Contact()
{
    ViewBag.Message = DateTime.Now.ToString();
    return View();
}
```

При запуске этого действия после добавления кеширования вывода вы только заметите, что значение `ViewBag.Message` изменяется раз в 60 секунд. Для дополнительного выяснения попробуйте поместить точку останова в метод контроллера. Вы увидите, что управление будет попадать в эту точку останова только при выполнении страницы в первый раз (когда кешированная версия еще не существует), а также каждый раз, когда срок хранения кешированной версии истекает.

Конфигурирование расположения кеша

Атрибут `OutputCacheAttribute` содержит ряд параметров, которые предоставляют полный контроль над тем, как и где кешируется контент страницы.

По умолчанию параметр `Location` установлен в `Any`, а это значит, что контент кешируется в трех расположениях: веб-сервер, любые прокси-серверы и веб-браузер пользователя. Параметр `Location` можно установить в любое из следующих значений: `Any`, `Client`, `Downstream`, `Server`, `None` или `ServerAndClient`.

Стандартное значение `Any` подходит для большинства сценариев, но временами возникают ситуации, когда необходим более точный контроль над тем, где кешируются данные.

Например, предположим, что нужно кешировать страницу, которая отображает имя текущего пользователя. Если использовать стандартное значение `Any`, то всем пользователям будет отображаться имя пользователя, который запросил страницу первым, что некорректно.

Во избежание этого сконфигурируйте кеш вывода со свойством `Location`, установленным в `OutputCacheLocation.Client`, и свойством `NoStore`, установленным в `true`, чтобы данные сохранялись только в веб-браузере пользователя:

```
[OutputCache(Duration = 3600, VaryByParam = "none", Location =
            OutputCacheLocation.Client, NoStore = true)]
public ActionResult About()
{
    ViewBag.Message = "The current user name is " + User.Identity.Name;
    return View();
}
```

Варьирование кеша вывода на основе параметров запроса

Один из наиболее мощных аспектов кеширования вывода связан с возможностью кешировать несколько версий одного и того же действия контроллера на основе параметров запроса, применяемых для вызова действия.

Например, пусть имеется действие контроллера по имени `Details`, который отображает детальные сведения об аукционном товаре:

```

public ActionResult Details(string id)
{
    var auction = _repository.Find<Auction>(id);
    return View("Details", auction);
}

```

В случае использования стандартной настройки кеширования вывода для каждого запроса будет отображаться детальные сведения об одном и том же аукционном товаре. Чтобы решить эту проблему, можно установить свойство `VaryByParam` для создания разных кешированных версий того же самого контента на основе параметра формы или строки запроса:

```

[OutputCache(Duration = int.MaxValue, VaryByParam = "id")]
public ActionResult Details(string id)
{
    var auction = _repository.Find<Auction>(id);
    return View("Details", auction);
}

```

Свойство `VaryByParam` предлагает довольно много опций, которые помогают указать, когда будет создаваться новая версия кеша. Если задано `"none"`, будет всегда получаться первая кешированная версия страницы. Если указано `"*"`, то всякий раз, когда значения параметров формы или строки запроса варьируются, будет создаваться другая кешированная версия. Разделяя записи, использующие строку запроса, можно определить список правил кеширования для параметров формы или строки запроса.

В табл. 12.2 представлен полный список свойств, доступных в `OutputCache Attribute`.

Таблица 12.2. Параметры кеширования вывода

Параметр	Описание
<code>CacheProfile</code>	Имя используемой политики кеша вывода
<code>Duration</code>	Промежуток времени в секундах для кеширования контента
<code>Enabled</code>	Включает/отключает кеш вывода для текущего контента
<code>Location</code>	Расположение кеша контента
<code>NoStore</code>	Включает/отключает HTTP-заголовок <code>Cache-Control</code>
<code>SqlDependency</code>	Пара, состоящая из имени базы данных и имени таблицы, от которых зависит элемент кеша
<code>VaryByContentEncoding</code>	Разделенный запятыми список наборов символов (кодировок контента), которые кеш вывода использует для изменения элементов кеша
<code>VaryByCustom</code>	Список специальных строк, которые кеш вывода использует для изменения элементов кеша
<code>VaryByHeader</code>	Разделенный запятыми список имен HTTP-заголовков, которые кеш вывода использует для изменения элементов кеша
<code>VaryByParam</code>	Разделенный точками с запятой список параметров формы или строки запроса, которые кеш вывода использует для изменения элементов кеша

Профили кеша вывода

Вместо снабжения атрибутом `OutputCacheAttribute` каждого действия контролера можно создать глобальные правила кеширования вывода за счет использования *профилей кеша вывода* в файле `web.config` приложения.

Тот факт, что профили кеша вывода находятся в одном месте, существенно упрощает настройку и сопровождение логики кеширования вывода сразу для всего сайта. В качестве дополнительной выгоды: для вступления в силу ни одно из этих изменений не требует перекомпиляции и повторного развертывания приложения.

Для применения профилей кеша вывода необходимо добавить раздел кеширования вывода в файл `web.config` приложения. Затем понадобится определить один или более профилей кеша и связанные с каждым из них параметры.

К примеру, показанный ниже профиль `ProductCache` кеширует контент страницы на протяжении часа и изменяет каждый кеш параметров запроса "id":

```
<caching>
    <outputCacheSettings>
        <outputCacheProfiles>
            <add name="ProductCache" duration="3600" varyByParam="id"/>
        </outputCacheProfiles>
    </outputCacheSettings>
</caching>
```

Этот профиль кеширования используется следующим образом:

```
[OutputCache(Duration = 0, VaryByParam = "none")]
public JsonResult Index()
{
    User user = new User { FirstName = "Joe", LastName = "Smith" };
    return Json(user);
}
```

"Кеширование бублика"

В сложном динамическом веб-приложении вы будете часто сталкиваться с необходимостью кешировать полную страницу, но продолжать генерацию определенных порций страницы. Например, в приложении ЕВиу имеет смысл кешировать большую часть страницы, но не порции, которые изменяются на основе вошедшего пользователя, такие как раздел ввода, отображающий имя текущего пользователя – очевидно, что все пользователи не должны видеть одно и то же пользовательское имя.

Если вам кажется, что решением будет применение `OutputCache` с `VaryByParam` и варьирование по идентификатору пользователя, подумайте еще раз. Атрибут `OutputCache` обеспечивает сохранение целой страницы, так что с помощью этого подхода можно будет сохранять каждый раз полную страницу для каждого пользователя с отличающимся именем (или чем-то другим, на чем основан динамический раздел). Таким образом, большая часть данных оказывается избыточной.

Именно в таких ситуациях в игру вступает *“кеширование бублика”* (*donut caching*). *“Кеширование бублика”* – это технология кеширования серверной стороны, при которой кешируется вся страница кроме небольших порций, остающихся динамическими. Эти небольшие порции похожи на “дыры” в кешированном контенте, что напоминает бублик и объясняет выбранное для технологии название.

Хотя механизм представлений Razor инфраструктуры ASP.NET MVC не имеет первоклассной поддержки *“кеширования бублика”*, ASP.NET Web Forms предлагает элемент управления `Substitution`, предназначенный для вырезания “дыр”, или организации динамических разделов:

```
<header>
    <h1>Donut Caching Demo</h1>
    <div class="userName">
        <asp:Substitution runat="server" MethodName="GetUserName" />
    </div>
</header>

<!-- Далее следует остальная часть страницы с кешируемым контентом. -->
```

Этот элемент управления регистрирует событие обратного вызова внутри кеша вывода ASP.NET, который затем вызывает статический метод на вашей странице, когда кешированная страница запрашивается:

```
partial class DonutCachingPage : System.Web.UI.MasterPage
{
    public static string GetUserName(HttpContext Context)
    {
        return "Hello " + Context.User.Identity.Name;
    }
}
```

Всякий раз, когда страница DonutCachingPage запрашивается, возвращается полная кешированная страница кроме раздела с именем пользователя, который продолжает динамически генерироваться для каждого запроса. Учитывая тот факт, что инфраструктура ASP.NET MVC построена поверх ASP.NET, мы можем пользоваться API-интерфейсами, которые элемент управления Substitution применяет для реализации функциональности, по поведению подобной ASP.NET MVC: класс HttpResponse имеет метод WriteSubstitution(), используемый элементом управления Substitution “за кулисами”.

С помощью этого метода можно написать специальный вспомогательный метод HTML для дублирования той же самой логики:

```
public delegate string CacheCallback(HttpContextBase context);
public static object Substitution(this HtmlHelper html, CacheCallback ccb) {
    html.ViewContext.HttpContext.Response.WriteSubstitution(
        c => HttpUtility.HtmlEncode(
            ccb(new HttpContextWrapper(c)))
    );
    return null;
}
```

Имея этот расширяющий метод, можно переписать предыдущий пример, чтобы применить новый вспомогательный метод:

```
<header>
    <h1>MVC Donut Caching Demo</h1>
    <div class="userName">
        Hello @Html.Substitution(context => context.User.Identity.Name)
    </div>
</header>

<!-- Далее следует остальная часть страницы с кешируемым контентом. -->
```

Теперь кешируется все представление, за исключением раздела внутри дескриптора `<div class="userName">`, привнося в ASP.NET MVC функциональность, похожую на предлагаемую элементом управления Substitution из Web Forms.

NuGet-пакет MvcDonutCaching

В рассмотренном выше примере демонстрировалась упрощенная версия “кэширования бублика”, которая не очень хорошо подходит для более сложных сценариев. Хотя “кэширование бублика” по-прежнему недоступно в составе ASP.NET MVC 4, пакет NuGet под названием MvcDonutCaching (<http://mvcdonutcaching.codeplex.com/>) может помочь легко реализовать более сложные сценарии. Этот пакет добавляет множество расширений к существующим вспомогательным методам HTML, а также добавляет специальный атрибут DonutOutputCacheAttribute, который может быть помещен на любое действие, нуждающееся в “кэшировании бублика”.

“Кэширование дырки от бублика”

“Кэширование дырки от бублика” (donut hole caching) является противоположностью “кэшированию бублика”: в то время как технология “кэширования бублика” кэширует целую страницу, оставляя лишь несколько небольших разделов, “кэширование дырки от бублика” кэширует только одну или несколько порций страницы (т.е. “дырки” от бублика).

Например, образцовое приложение Ebucу содержит список категорий аукционных товаров, которые изменяются нечасто, поэтому имеет смысл визуализировать все категории один раз и кэшировать результирующий HTML-контент.

“Кэширование дырки от бублика” очень удобно в сценариях такого вида, когда большинство элементов на странице являются динамическими, за исключением нескольких разделов, которые изменяются редко либо изменяются на основе параметра запроса. И, в отличие от “кэширования бублика”, ASP.NET MVC предлагает великолепную поддержку “кэширования дырки от бублика” через использование дочерних действий.

Давайте посмотрим на “кэширование дырки от бублика” в работе, применив его к упомянутому ранее примеру с категориями аукционных товаров в приложении Ebucу. Ниже приведено частичное представление, которое будет кэшироваться:

```
@{  
    Layout = null;  
}  
  
<ul>  
    @foreach(var category in ViewBag.Categories as IEnumerable<Category>)  
        <li>@Html.ActionLink(@category.Name, "category", "categories",  
                           new { categoryId = category.Id })</li>  
    </ul>
```

Это частичное представление проходит в цикле по всем категориям и визуализирует каждую в виде элемента неупорядоченного списка. Каждый элемент в этом списке является ссылкой на действие Category в контроллере Categories с передачей в качестве параметра действия идентификатора (Id) категории.

Далее создадим дочернее действие, которое отображает это представление:

```
[ChildActionOnly]  
[OutputCache(Duration=60)]  
public ActionResult CategoriesChildAction()  
{  
    // Извлечь категории из базы данных и передать  
    // их дочернему представлению через ViewBag.  
    ViewBag.Categories = Model.GetCategories();  
    return View();  
}
```

Обратите внимание, что атрибут `OutputCacheAttribute` кеширует результат этого метода на протяжении 60 секунд. Затем это новое действие можно вызвать в родительском представлении посредством `@Html.Action("CategoriesChildAction")`, как показано в следующем примере:

```
<header>
    <h1>MVC Donut Hole Caching Demo</h1>
</header>
<aside>
    <section id="categories">
        @Html.Action("CategoriesChildAction")
    </section>
</aside>
<!-- Далее следует остальная часть страницы с некешируемым контентом. -->
```

Теперь, когда страница визуализируется, вызывается дочернее действие `Categories` для генерации списка категорий.

Результаты этого вызова кешируются через `OutputCacheAttribute`, поэтому при визуализации этой страницы в следующий раз список `Categories` берется из кеша, а остальная часть страницы генерируется с нуля.

Распределенное кеширование

В случаях, когда множество экземпляров приложения выполняются на более чем одном веб-сервере, запросы к приложению могут обслуживаться любым из этих серверов. И каждый раз, когда запрос направляется на новый сервер, кешированные элементы должны быть повторно сгенерированы, если это еще не делалось на данном сервере.

В зависимости от сложности процесса генерации кешированных элементов, может оказаться крайне неэффективно повторно генерировать одни и те же данные снова и снова. Вместо этого намного более практично сгенерировать данные однажды и сохранить их на множестве серверов или веб-ферм. Эта технология кеширования данных в одном экземпляре приложения и разделения их с остальными экземплярами известна под названием *распределенного кеширования*, и она является наиболее сложной из всех технологий кеширования.

Распределенное кеширование – это расширение нормальных технологий кеширования, когда информация из базы данных или сеанса хранится в центральном расположении, к которому имеют доступ все экземпляры приложения. Ниже перечислены преимущества использования уровня распределенного кеширования.

- **Производительность.** Поскольку большой объем данных может быть сохранен в памяти серверов, существенно улучшается производительность чтения, что позволяет страницам загружаться быстрее.
- **Масштабируемость.** Масштабируемость становится функцией добавления дополнительной мощности или узлов к кластерам, позволяя легко масштабировать приложение до более высоких требований и показателей нагрузки. В сочетании с облачным хранилищем данных узлы можно выделять по требованию и освобождать, когда они не нужны, повышая эффективность затрат.
- **Избыточность.** Избыточность гарантирует, что если один узел или сервер отказывает, приложение в целом не пострадает. Вместо этого другой узел обхода отказа просто подхватит запрос и обслужит его безо всякого ручного вмешательства. Обход отказа и избыточность – это важные функциональные средства многих решений распределенного кеширования.

Решения распределенного кеширования

На сегодняшний день доступно несколько продуктов для распределенного кеширования, и хотя каждый из них предлагает довольно отличающиеся API-интерфейсы и способы работы с данными, которыми управляют эти продукты, базовые концепции распределенного кеширования остаются теми же самыми. Чтобы представить идею реализации решения распределенного кеширования, в следующих разделах будет показано, как внедрить решение распределенного кеширования от Microsoft, которое называется *Velocity*, в рамках образцового приложения *Ebuy*.

Установка Velocity. Продукт *Velocity* – это уровень кеширования Windows AppFabric (также называемый Microsoft Application Server). Таким образом, чтобы установить *Velocity*, потребуется загрузить продукт Windows AppFabric (<http://msdn.com/appfabric>) или установить его через Web Platform Installer (<http://www.microsoft.com/web/downloads/platform.aspx>).

После запуска установщика и получения страницы Feature Selection (Выбор средств), показанной на рис. 12.1, выберите средства Caching Services (Службы кеширования) и Cache Administration (Администрирование кеша), отметив соответствующие флажки. Если вы работаете в среде Windows 7, установите расширение IIS 7 Manager for Remote Administration (<http://www.iis.net/download/IISManager>), которое позволит управлять удаленными серверами IIS из машины Windows 7.



Если вы хотите использовать только кеширующую часть, выполните автоматическую установку (<http://msdn.microsoft.com/ru-ru/library/ff637714.aspx>) или введите команду `SETUP /i CACHINGSERVICE`.

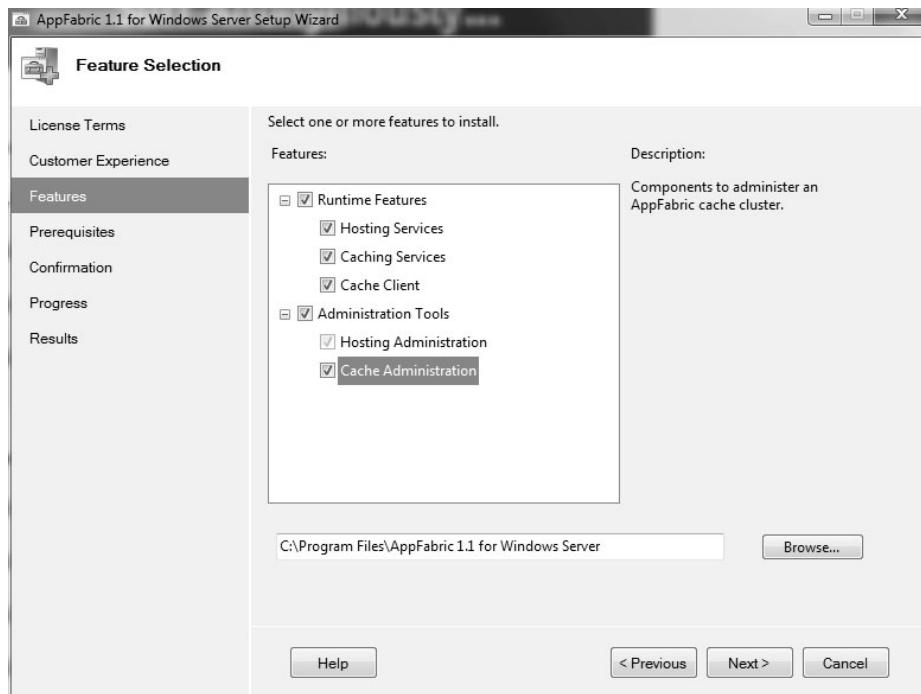


Рис. 12.1. Страница Feature Selection установщика Windows AppFabric

После установки Windows AppFabric отобразится мастер конфигурирования, который поможет пройти по оставшейся части процесса, начиная с указания места сохранения конфигурационной информации Velocity.

В рассматриваемом примере выберите вариант с базой данных и щелкните на кнопке Next (Далее) для перехода к конфигурированию параметров базы данных (рис. 12.2).

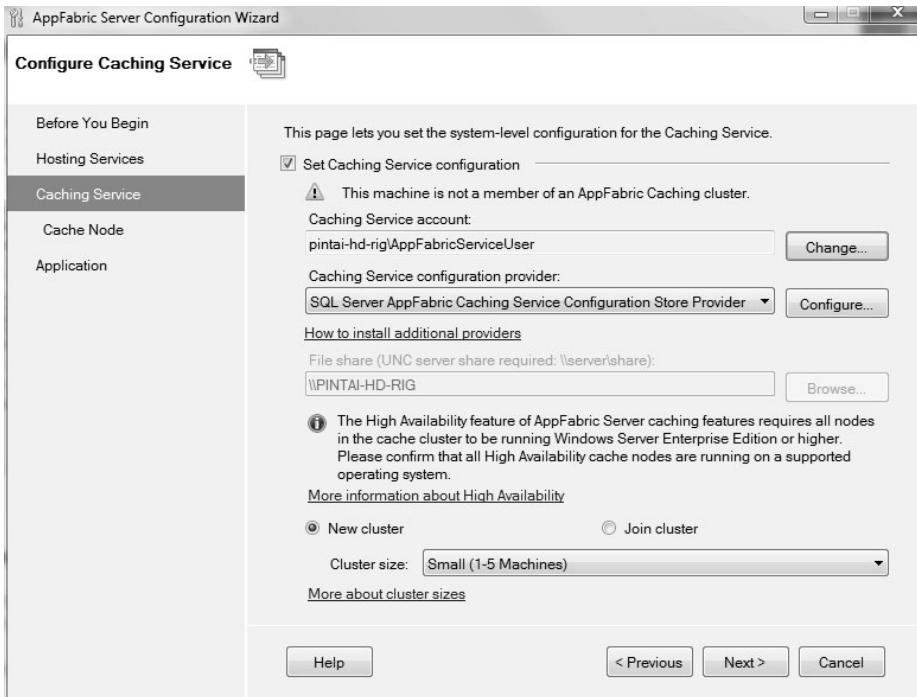


Рис. 12.2. Конфигурирование параметров базы данных

Администрирование кластера памяти из PowerShell. Следующий шаг заключается в администрировании кеша с применением PowerShell. К этому моменту в меню All Programs (Все программы) должен появиться новый пункт под названием Caching Administration Windows PowerShell (Администрирование кешированием с помощью Windows PowerShell).

Используя эту консоль, можно управлять кешами, проверять их активность и создавать новые кеши.

Перед тем как начать, понадобится “запустить” кластер кеша. Введите в консоли PowerShell приведенную ниже команду:

```
C:\> Start-CacheCluster
```

Затем введите следующую команду, чтобы выдать своей пользовательской учетной записи права на доступ к кластеру кеша:

```
C:\> Grant-CacheAllowedClientAccount 'домен\имя_пользователя'
```

Чтобы проверить, получила ли пользовательская учетная запись права на доступ, воспользуйтесь командой Get-CacheAllowedClientAccounts.



Для просмотра всех связанных с кешем команд применяйте команду
get-command *cache*.

Использование кеша. К кешу можно привязаться в файле web.config или в коде. Ниже приведен пример кода, в котором это делается вручную внутри вспомогательного метода:

```
using Microsoft.ApplicationServer.Caching;
using System.Collections.Generic;

public class CacheUtil
{
    private static DataCacheFactory _factory = null;
    private static DataCache _cache = null;

    public static DataCache GetCache()
    {
        if (_cache != null)
            return _cache;

        // Определить массив для одного хоста кеша.
        List<DataCacheServerEndpoint> servers = new List<DataCacheServerEndpoint>(1);

        // Указать подробные сведения для хоста кеша:
        // параметр 1 = имя хоста
        // параметр 2 = номер порта кеша
        servers.Add(new DataCacheServerEndpoint("mymachine", 22233));

        // Создать конфигурацию кеша.
        DataCacheFactoryConfiguration configuration =
            new DataCacheFactoryConfiguration();

        // Установить хост(ы) кеша.
        configuration.Servers = servers;

        // Установить стандартные свойства для локального кеша
        // (локальный кеш отключен).
        configuration.LocalCacheProperties =
            new DataCacheLocalCacheProperties();

        // Отключить трассировку во избежание получения
        // информационных/многословных сообщений на веб-странице.
        DataCacheClientLogManager.ChangeLogLevel(System.Diagnostics.TraceLevel.Off);

        // Передать конфигурационные настройки конструктору DataCacheFactory.
        _factory = new DataCacheFactory(configuration);

        // Получить ссылку на именованный кеш "default".
        _cache = _factory.GetCache("default");

        return _cache;
    }
}
```

После того как кеш настроен, пользоваться им довольно легко. Вот как добавить элемент к объекту кеша, созданному в предыдущем листинге:

```
var cache = CacheUtil.GetCache();
cache.Add(orderid, order);
```

Извлечение кэшированного элемента также осуществляется просто:

```
Order order = (Order)cache.Get(orderid);
```

Не менее просто производится и обновление существующего элемента:

```
cache.Put(orderid, order);
```

Можно также заменить стандартный поставщик состояния сеанса кешированием AppFabric.

Ниже приведен пример содержимого файла web.config:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <!-- configSections должен быть ПЕРВЫМ элементом -->
    <configSections>
        <!-- Требуется для чтения элемента <dataCacheClient> -->
        <section name="dataCacheClient"
            type=
                "Microsoft.ApplicationServer.Caching.DataCacheClientSection,
                Microsoft.ApplicationServer.Caching.Core, Version=1.0.0.0,
                Culture=neutral, PublicKeyToken=31bf3856ad364e35"
            allowLocation="true"
            allowDefinition="Everywhere"/>
    </configSections>
    <!-- Клиент кеша -->
    <dataCacheClient>
        <!-- Хост(ы) кеша -->
        <hosts>
            <host
                name="CacheServer1"
                cachePort="22233"/>
        </hosts>
    </dataCacheClient>
    <system.web>
        <sessionState mode="Custom"
            customProvider="AppFabricCacheSessionStoreProvider">
            <providers>
                <!-- Указать именованный кеш для данных сеанса -->
                <add
                    name="AppFabricCacheSessionStoreProvider"
                    type=
                        "Microsoft.ApplicationServer.Caching.DataCacheSessionStoreProvider"
                    cacheName="NamedCache1"
                    sharedId="SharedApp"/>
            </providers>
        </sessionState>
    </system.web>
</configuration>
```

Как видите, AppFabric предлагает продуманные функциональные средства, которые позволяют этому продукту охватывать диапазон от простого механизма кеширования до замены стандартного поставщика состояния сеанса.

Дополнительные сведения о концепциях, возможностях и архитектуре AppFabric доступные на MSDN-странице по адресу [http://msdn.microsoft.com/ru-ru/library/ff383731\(v=azure.10\).](http://msdn.microsoft.com/ru-ru/library/ff383731(v=azure.10).)

Технологии кеширования клиентской стороны

Браузеры отображают веб-страницы пользователю путем извлечения HTML-разметки, данных и ресурсов поддержки, таких как файлы CSS, изображений, JavaScript-кода, cookie-наборов, роликов Flash и т.д. Однако вне зависимости от того, насколько скоростным является пользовательское подключение к Интернету, всегда быстрее отобразить что-либо, извлекая его из жесткого диска пользователя, а не передавая через Интернет.

Проектировщикам браузеров это известно, поэтому, где только возможно, они за-действуют кеширование для сохранения ресурсов на диске, тем самым избегая доступа в сеть. Процесс прост: в любой момент, когда производится обращение к веб-странице, браузер проверяет жесткий диск на предмет наличия локальных копий любых файлов, являющихся частью этой страницы. В случае их отсутствия браузер загружает необходимые ресурсы из сервера. При первом посещении страницы это касается всего. При последующих визитах браузер должен загружать страницу быстрее, с учетом того, что он уже имеет локальный доступ к ресурсам, которые требуются для отображения страницы (если только страница не запрашивает новые или отличающиеся ресурсы).

Браузер кеширует ресурсы, сохраняя их локально на жестком диске в предваритель-но выделенной области заданного размера. Пользователи могут управлять объемом дискового пространства, выделяемого под это хранилище. Браузер заботится об очист-ке старых элементов и управлении ресурсами без какого-либо вмешательства или ввода со стороны пользователя. А теперь давайте посмотрим, как можно использовать кеши-рование клиентской стороны или кеш браузера для ускорения работы приложения.

Кеш браузера

Ресурсы, кешируемые локально браузером, управляются тремя базовыми механиз-мами: *объявление новым*, *проверка достоверности* и *объявление недействительным*. Они яв-ляются частью самого протокола HTTP и определены HTTP-заголовками.

Механизм объявления новым позволяет применять ответ без повторной проверки на сервере-источнике и может управляться как сервером, так и клиентом. Например, заголовок Expires указывает дату, когда документ станет устаревшим, а директива Cache-Control: max-age сообщает кешу, сколько секунд ответ считается новым.

В следующем примере показано, как устанавливать эти заголовки с помощью кода серверной стороны:

```
public ActionResult CacheDemo()
{
    // Установить заголовок Cache-Control в одно из значений
    // перечисления HttpCacheability.
    // См. http://msdn.microsoft.com/en-us/library/system.web.
    // httpcacheability(v=vs.110).aspx
    // Установить заголовок Cache-Control: public для указания, что ответ
    // является кешируемым клиентами и разделяемыми (прокси) кешами.
    Response.Cache.SetCacheability(HttpCacheability.Public);
    // Установить заголовок Cache-Control: max-age в 20 минут.
    Response.Cache.SetMaxAge(DateTime.Now.AddMinutes(20));
    // Установить заголовок Expires в 11:00 Р.М. локального времени
    // текущего дня истечения.
    Response.Cache.SetExpires(DateTime.Parse("11:00:00PM"));
    return View();
}
```

Проверка достоверности используется для выяснения, является ли кешированный ответ по-прежнему подходящим после того, как он устарел. Например, если ответ имеет заголовок `Last-Modified`, кеш может делать условный запрос с применением заголовка `If-Modified-Since`, чтобы посмотреть, изменился ли контент. Тем не менее, это довольно слабая форма проверки достоверности; если нужна более строгая проверка, можно воспользоваться механизмом `ETag` (тег сущности).

В следующем коде демонстрируются оба подхода:

```
public ActionResult CacheDemo()  
{  
    // Установить HTTP-заголовок Last-Modified в указанное значение  
    // даты и времени.  
    Response.Cache.SetLastModified(DateTime.Parse("1/1/2012 00:00:01AM"));  
  
    // Установить HTTP-заголовок ETag в указанную строку.  
    Response.Cache.SetETag("\"someuniquestring:version\"");  
  
    return View();  
}
```



Активизация механизма *объявления недействительным* обычно является побочным эффектом действия другого запроса, который проходит через кеш. Скажем, если по URL, ассоциированному с кешированным ответом, поступает запрос POST, PUT или DELETE, кешированный ответ будет объявлен недействительным.

Хотя кеш браузера является хорошей возможностью обеспечить более быструю загрузку страниц, с ним связано множество проблем, накопленных с годами. Ошибки, проблемы безопасности и отсутствие детализированного контроля над тем, что кешируется, создает множество сложностей для веб-разработчиков. Кроме того, невозможность объявить недействительным кеш, когда элемент на сервере изменился, часто требует реализации специальных низкоуровневых приемов, которые в конечном итоге приводят к запутанному коду.

Новая спецификация HTML 5 помогает устраниить некоторые из упомянутых проблем, предоставляя разработчикам новые технологии и более детальный контроль над кешем клиентской стороны.

API-интерфейс ApplicationCache

В спецификации HTML 5 определен API-интерфейс `ApplicationCache` (<http://www.whatwg.org/specs/web-apps/current-work/#applicationcache>), который обеспечивает разработчикам прямой доступ в локальный кеш контента браузера.

Чтобы включить `ApplicationCache` в своем приложении, понадобится выполнить следующие шаги.

1. Определение манифеста.
2. Ссылка на манифест.
3. Использование манифеста.

Рассмотрим реализацию всех этих шагов более подробно.

Определение манифеста

Определение файла манифеста сводится к созданию текстового файла с расширением .manifest:

```
CACHE MANIFEST
# Версия 0.1
home.html
site.css
application.js
logo.jpg
```

Это простой файл манифеста, который сообщает браузеру о необходимости кеширования указанных в манифесте четырех файлов. В первой строке файла должен содержаться текст CACHE MANIFEST.

Ниже приведен чуть более сложный пример, в котором демонстрируется возможность детализированного контроля над тем, что кешируется:

```
CACHE MANIFEST
# Сгенерирован 04-23-2012:v2
# Кешированные элементы.
CACHE:
/favicon.ico
home.html
site.css
images/logo.jpg
scripts/application.js

# Ресурсы, которые "всегда" извлекаются из сервера
NETWORK:
login.asmx

# Использовать index.html (статическая версия домашней страницы) ,
# если /Home/Index не доступен
# Использовать offline.jpg вместо всех изображений в папке images/
# Использовать appOffline.html вместо всех других маршрутов
FALLBACK:
/Home/Index /index.html
images/ images/offline.jpg
* /appOffline.html
```

Легко заметить, что в файле манифеста задействовано несколько базовых соглашений.

- Строки, начинающиеся с символа #, являются комментариями.
- В разделе CACHE перечислены ресурсы, которые будут кешироваться после доступа к веб-сайту в первый раз.
- В разделе NETWORK перечислены ресурсы, которые браузер должен всегда извлекать из сервера – другими словами, эти ресурсы никогда не кешируются.
- В разделе FALLBACK определены ресурсы, которые будут использоваться в случае, если соответствующие ресурсы недоступны или заняты. Это является совершенно не обязательным, и здесь поддерживаются групповые символы.

Следующий шаг заключается в сообщении браузеру об этом файле манифеста для приложения.

Ссылка на манифест

Чтобы сослаться на манифест, просто определите атрибут `manifest` в дескрипторе `<html>`:

```
<!DOCTYPE html>
<html manifest="site.manifest">
...
</html>
```

Благодаря атрибуту `manifest`, браузер выясняет, что приложение определяет манифест кеша, и пытается загрузить файл манифеста автоматически.

Корректное использование манифеста

Ключ к применению манифеста состоит в его использовании с корректным MIME-типов (`text/cache-manifest`):

```
Response.ContentType = "text/cache-manifest";
```

Если не указать этот MIME-тип, браузер не сможет распознать этот файл как файл манифеста, и средство `ApplicationCache` не будет доступно для сайта.

При включенном `ApplicationCache` для приложения браузер будет извлекать ресурсы только в следующих трех случаях.

1. Когда пользователь очищает кеш, что приводит к удалению всего кешированного контента.
2. Когда файл манифеста изменяется на сервере. Простая модификация комментария и сохранение этого файла может инициировать обновление.
3. Когда кеш обновляется программно посредством кода JavaScript.

Как видите, API-интерфейс `ApplicationCache` предоставляет полный контроль над тем, что кешируется, и позволяет инициировать обновления, когда это необходимо, не прибегая к каким-либо обходным маневрам или низкоуровневому коду.

В следующем разделе будет описано еще одно средство из спецификации HTML 5, которое дает возможность кешировать элементы в браузере отличающимся от `ApplicationCache` образом.

Локальное хранилище

Еще одним новым средством, появившимся в спецификации HTML 5, является поддержка оффлайнового механизма хранения на основе браузера, которое называется *локальным хранилищем* (Local Storage). Локальное хранилище можно считать “супер-cookie-набором”, не ограниченным размерами обычных cookie-наборов браузера: оно позволяет сохранять большие объемы данных на устройстве пользователя.

API-интерфейс `LocalStorage` состоит из двух конечных точек для управления локальным хранилищем данных: `localStorage` и `sessionStorage`. Хотя объекты `localStorage` и `sessionStorage` предлагают похожие методы, основное отличие между ними заключается в том, что данные, хранящиеся в `localStorage`, доступны неопределенно долго, тогда как данные, хранящиеся в `sessionStorage`, очищаются при закрытии страницы в браузере.



Подобно большинству объектов кеширования серверной стороны, локальное хранилище использует структуру словаря данных на основе строк.

Таким образом, в случае извлечения данных, отличных от строк, может понадобиться применение функций `parseInt()` или `parseFloat()` для приведения данных обратно к встроенным типам данных JavaScript.

Для сохранения элемента в `localStorage` можно использовать `setItem()`:

```
localStorage.setItem("userName", "john");
localStorage.setItem("age", 32);
```

Или же можно применить синтаксис с квадратными скобками:

```
localStorage[userName] = "john";
localStorage["age"] = 32;
```

Извлечение элемента также осуществляется просто:

```
var userName = localStorage.getItem("userName");
var age = parseInt(localStorage.getItem("age"));

// Или синтаксис с квадратными скобками...
var userName = localStorage["userName"];
var age = parseInt(localStorage["age"]);
```

С помощью функции `removeItem()` из хранилища удаляется любой отдельный элемент:

```
localStorage.removeItem("userName");
```

А вот как очистить все ключи за один раз:

```
localStorage.clear();
```

Объем памяти, выделенной под локальное хранилище, не является бесконечным; существует предел того, сколько в нем можно сохранить. В черновой спецификации этот предел произвольно установлен равным 5 Мбайт, хотя браузеры могут реализовать и большие пределы. Приложение может запросить больше памяти, что в результате приводит к отображению пользователю запроса. Это заставляет пользователя принять решение относительно того, разрешать ли выделение дополнительной памяти под хранилище.

Локальное хранилище предлагает API-интерфейсы для определения того, сколько пространства осталось в рамках квоты, и для запрашивания дополнительной памяти. Метод `localStorage.remainingSpace()` возвращает объем оставшегося дискового пространства в байтах. Когда объем сохраненных данных превышает установленный предел, браузер может генерировать исключение `QuotaExceededError` или запросить у пользователя разрешение на выделение дополнительной памяти.

Большинство современных браузеров поддерживают `localStorage`; однако, не взирая ни на что, всегда рекомендуется выяснить, существует ли отдельная функциональная возможность в браузере, прежде чем использовать ее. В следующем фрагменте кода показано, как проверить, поддерживается ли объект `localStorage` в текущем браузере:

```
function IsLocalStorageSupported() {
    try {
        return 'localStorage' in window && window['localStorage'] !== null;
    } catch (e) {
        return false;
    }
}
```

Резюме

Кеширование является важным аспектом при построении масштабируемых и высокопроизводительных приложений. В этой главе были описаны различные технологии кеширования и представлены практические сценарии их использования. Помимо встроенных механизмов кеширования, таких как `HttpContext.Application`, `HttpContext.Session` и `OutputCache`, можно также применять уровень распределенного кеширования, чтобы достичь большей эффективности. Технологии “кеширования бублика” и “кеширования дырки от бублика” предлагают ряд интересных вариаций обычного кеширования и в подходящих сценариях могут оказаться весьма эффективными. Наконец, кеширование не ограничивается серверной стороной. Благодаря новым средствам, добавленным в HTML 5, идея кеширования может быть расширена также на клиентскую сторону. Спецификация HTML 5 привносит два новых очень гибких механизма для включения кеширования клиентской стороны и кеширования в браузере, обеспечивая прямой контроль над кешируемыми данными. Кроме того, новые опции хранилища клиентской стороны, доступные в рамках спецификации HTML 5, значительно упрощают поддержку “оффлайнового” режима, позволяя приложению продолжать работу в отсутствие активного подключения к Интернету.

Технологии оптимизации клиентской стороны

Конечная цель относительно производительности у любого разработчика заключается в том, чтобы обеспечить максимально быструю загрузку веб-страницы. Чем быстрее загружается страница, тем более отзывчивым выглядит сайт и тем комфортнее чувствуют себя его пользователи. Под *оптимизацией клиентской стороны* в целом понимается набор технологий, которые могут помочь в сокращении времени загрузки страниц.

В этой главе рассматривается ряд базовых технологий, дающих максимальную отдачу от потраченных средств. Хотя и нет какой-то одной технологии, подходящей на все случаи жизни, соблюдение изложенных в главе правил должно помочь любой довольно хорошо спроектированной странице достигнуть наилучших показателей в плане своего времени загрузки.

Большинство представленных здесь технологий не будут требовать переписывания существенного объема кода; вместо этого они могут применяться к любому хорошо спроектированному приложению за пределами кода.

Для чего заниматься оптимизацией? Страница, которая загружается быстро, также выглядит более отзывчивой в сценариях с ограниченной пропускной способностью сети. Если целевая аудитория работает в медленной сети, оптимизация поможет даже больше, обеспечив более компактную страницу, которая пользователь может увидеть гораздо быстрее.

Структура страницы

Чтобы разобраться, какие аспекты влияют на время загрузки страницы, давайте взглянем, каким образом браузер визуализирует страницу. Веб-страница главным образом состоит из HTML-разметки, файлов JavaScript и таблиц стилей, но также содержит изображения и возможно другие медиаданные, такие как объекты Flash или Silverlight.

При визуализации веб-страницы браузеры следуют нисходящему подходу: они начинают с верхней части HTML-разметки и запускают загрузку ресурсов по мере их появления в разметке (рис. 13.1). Страница не визуализируется и не отображается до тех пор, пока все ресурсы не будут загружены. Это означает, что даже если вся HTML-разметка страницы загружена, пользователи по-прежнему будут видеть пустой экран, пока браузер не завершит загрузку других ресурсов, находящихся на странице (вроде изображений, таблиц стилей и файлов JavaScript).

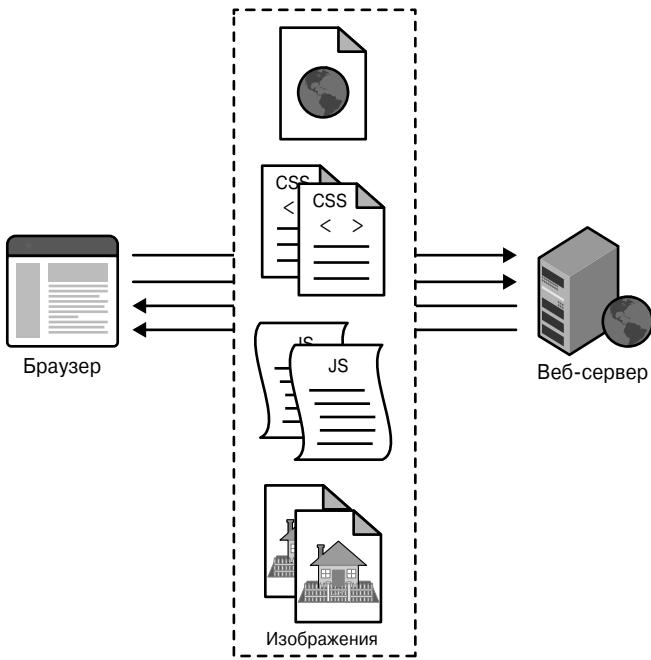


Рис. 13.1. Структура страницы

Подведем итоги:

- наличие меньшего количества ресурсов снижает время загрузки страницы;
- перестановка ресурсов на странице может привести к изменению момента, когда страница (или ее часть) отображается.

Структура HTTP-запроса

Большее число запросов делают страницу медленнее. Но почему? Давайте посмотрим, что происходит при запросе ресурса, чтобы понять, какие факторы влияют на время загрузки. Ниже перечислены соответствующие шаги.

1. *Поиск в DNS.* Первый шаг заключается в преобразовании доменного имени для запроса:
 - сначала браузер или клиент отправляет DNS-запрос DNS-серверу локального поставщика Интернет-услуг;
 - затем DNS-сервер отвечает IP-адресом для заданного имени хоста.
2. *Подключение.* Клиент устанавливает TCP-подключение с IP-адресом имени хоста.
3. *Выдача HTTP-запроса.* Браузер отправляет HTTP-запрос веб-серверу.
4. *Ожидание.* Затем браузер ожидает поступления ответа на запрос от веб-сервера:
 - на стороне сервера веб-сервер обрабатывает запрос, что включает поиск ресурса, и отправляет ответ клиенту;
 - браузер получает от веб-сервера первый байт первого пакета, который содержит HTTP-заголовки и контент.
5. *Загрузка.* Браузер загружает контент ответа.

6. *Закрытие*. После получения последнего байта браузер запрашивает у сервера закрытие подключения.

Эти шаги (показанные на рис. 13.2) повторяются для каждого запроса, который пока еще не находится в кеше браузера. Если запрошенный ресурс присутствует в кеше браузера, браузер просто загружает этот ресурс из кеша и не обращается к серверу с целью его загрузки оттуда. После загрузки ресурса из сервера браузер также пытается его кэшировать локально.

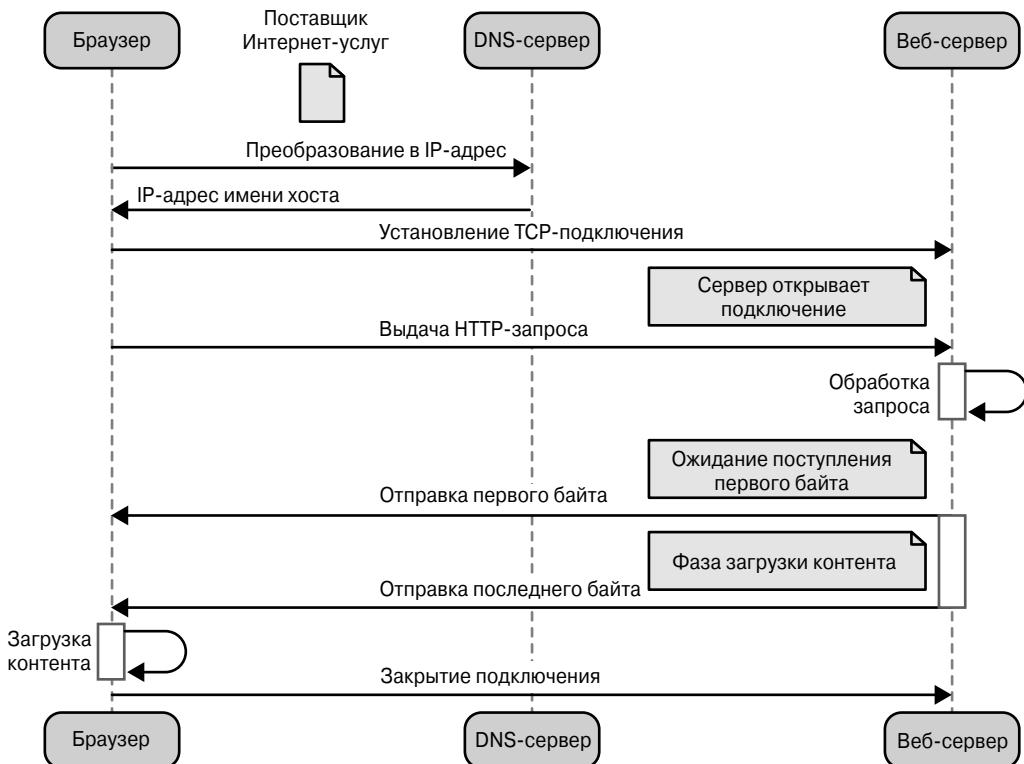


Рис. 13.2. Структура HTTP-запроса

Оптимизация частей этих шагов может помочь снизить время получения ответа.

Рекомендуемые приемы

Команда Exceptional Performance (Исключительная производительность) в Yahoo! идентифицировала 35 рекомендуемых приемов, которые помогают улучшить производительность веб-страниц. Они начали с 13 простых правил и затем расширили набор правил до 35, разделив их на 7 категорий. Полный список можно найти в блоге этой команды по адресу <http://developer.yahoo.com/performance/rules.html>.

В Google также имеется собственный набор правил улучшения производительности веб-сайтов. Рекомендации Google (https://developers.google.com/speed/docs/best-practices/rules_intro) включают свыше 30 правил, которые разделены на 6 категорий.

В нескольких следующих разделах будет представлена краткая сводка по ряду базовых правил, которые могут помочь в создании исключительно отзывчивых веб-сайтов.

Делайте меньше HTTP-запросов

Около 80% времени на ответы конечных пользователей приходится на интерфейсную часть. Большинство этого времени тратится на загрузку компонентов страницы, таких как изображения, таблицы стилей, сценарии, ролики Flash и т.д. Сокращение количества компонентов, в свою очередь, снижает число HTTP-запросов (рис. 13.3). Это ключевой момент для получения более быстрых страниц.

Можно перепроектировать страницу, чтобы сократить количество находящихся на ней компонентов, или объединить ряд внешних ресурсов (файлов JavaScript, таблиц стилей, изображений) для уменьшения числа компонентов, которые должны загружаться на сторону клиента. Инфраструктура ASP.NET MVC 4 предлагает готовую возможность “пакетирования”, которая помогает скомбинировать несколько файлов JavaScript и таблиц стилей в “пакетированный” ресурс, сокращающий количество загружаемых компонентов. Это средство подробно рассматривается в разделе “Пакетирование и минимизация” далее в главе.

Указанная технология уменьшает число запросов к сценариям и таблицам стилей, однако она не может быть применена к изображениям. Если страница полна изображений, подумайте об использовании технологии спрайтов CSS (CSS Sprites; <http://www.alistapart.com/articles/sprites>), которая позволяет сократить количество запросов изображений. Еще одна технология предполагает *встраивание* изображений или данных, представляющих изображения, в страницу или таблицу стилей с применением схемы URL вида data: (<http://tools.ietf.org/html/rfc2397>), но следует иметь в виду, что это не поддерживается всеми основными браузерами (http://en.wikipedia.org/wiki/Data_URI_scheme#Web_browser_support; http://ru.wikipedia.org/wiki/Data:_URL).

Используйте сеть доставки контента

Сеть доставки контента (content delivery network – CDN) – это коллекция веб-серверов, распределенных по множеству расположений с целью более эффективной доставки контента пользователям. Сервер для доставки контента конкретному пользователю обычно выбирается на основе сетевой близости: предпочтение отдается серверу с наименьшим количеством прыжков либо обладающему самым быстрым откликом.

Переход на CDN является относительно простым изменением, которое радикально улучшает показатели скорости веб-сайта. Кроме того, можно максимизировать количество загружаемых изображений, таблиц стилей и сценариев за счет применения нескольких субдоменов на сервере или в CDN. Браузер ограничивает число подключений для загрузки ресурсов, приходящееся на домен. За счет разнесения ресурсов по множеству субдоменов можно существенно увеличить количество параллельных загрузок, поскольку браузер трактует их как отдельные домены и использует больше подключений для разных доменов (рис. 13.4).

Добавляйте заголовок Expires или Cache-Control

Согласно исследованиям (<http://yuiblog.com/blog/2007/01/04/performance-research-part-2/>), 40–60% ежедневных посетителей сайта заходят на него с пустым кешем.

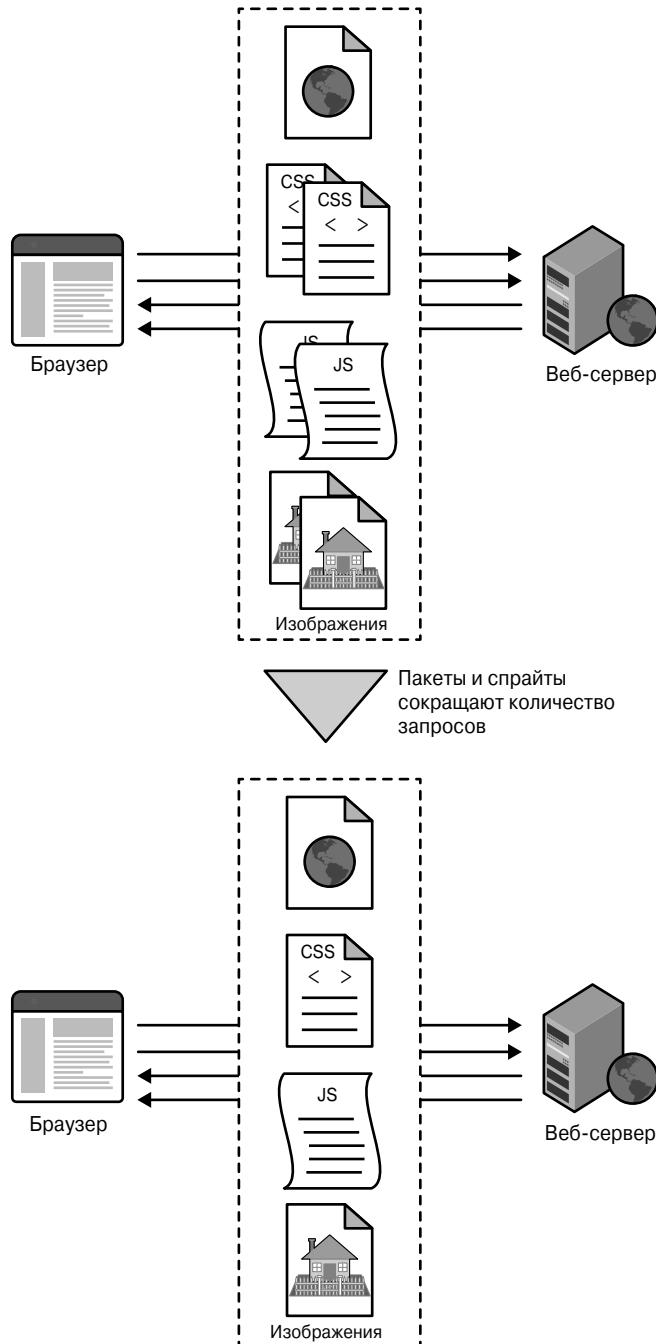


Рис. 13.3. Меньшее количество HTTP-запросов способствует более быстрой загрузке страницы

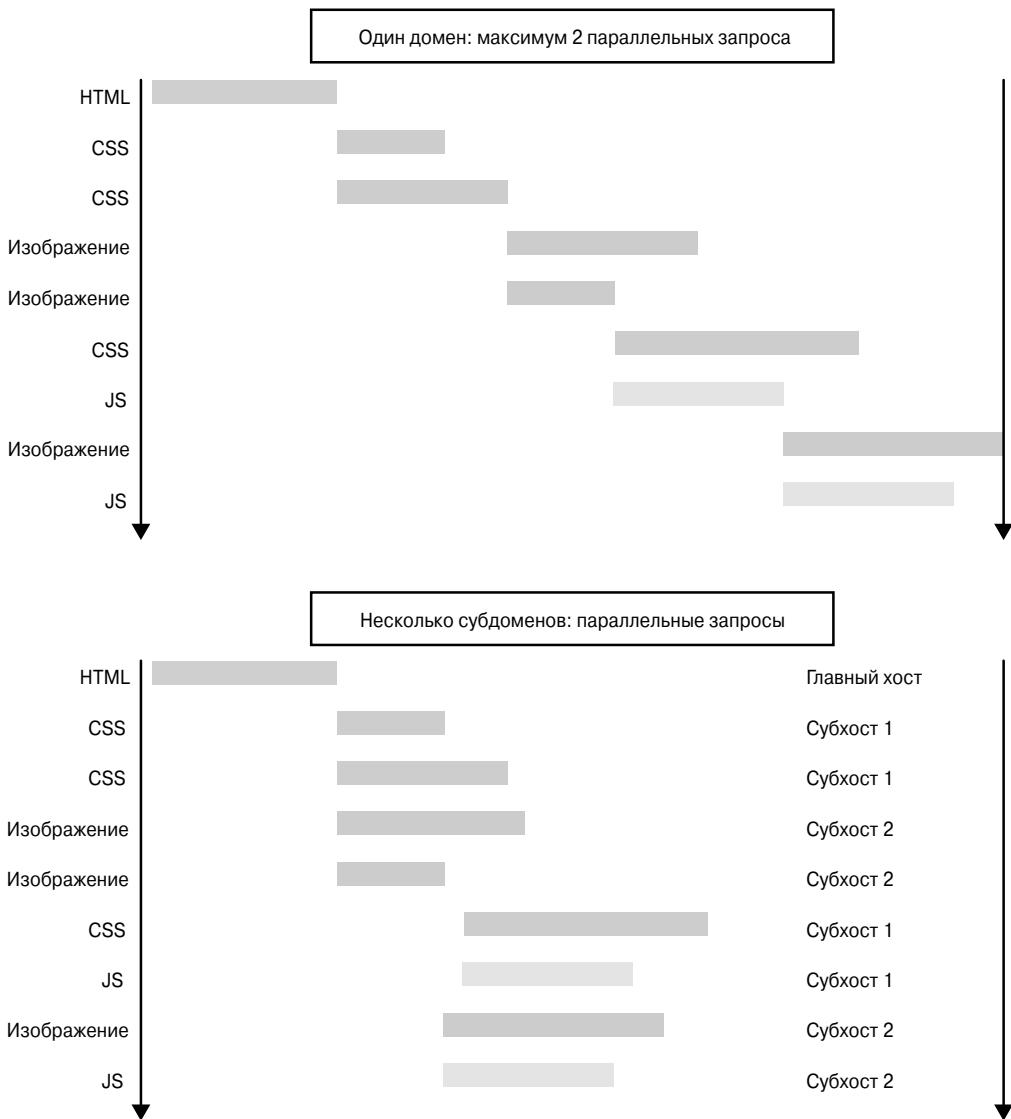


Рис. 13.4. Использование CDN

Предшествующие приемы (уменьшение количества HTTP-запросов, использование CDN) помогали ускорять первое обращение к страницам, а применение заголовка Expires или Cache-Control обеспечивает ускорение последующих посещений страниц, включая кеширование на стороне клиента.

Браузеры (и прокси-серверы) используют кеш с целью сокращения количества и размеров HTTP-запросов, делая загрузку веб-страниц быстрее. Веб-сервер использует заголовок Expires в HTTP-ответе для сообщения клиенту о том, насколько долго компонент может быть кеширован. Ниже показан заголовок Expires, который указывает браузеру на то, что этот ответ не устареет вплоть до 20 мая 2013 г.:

```
Expires: Mon, 20 May 2013 20:00:00 GMT
```

Добавлять заголовки Expires и Cache-Control можно в IIS или программно посредством ASP.NET MVC.

Настройка клиентского кеширования в IIS

Веб-сервер IIS 7 позволяет настраивать заголовки клиентского кеширования (<http://www.iis.net/ConfigReference/system.webServer/staticContent/clientCache>) с помощью подэлемента `<clientCache>` элемента `<staticContent>`.

Атрибут `httpExpires` добавляет HTTP-заголовок `Expires`, который задает дату и время, когда истекает срок хранения контента. Заголовки `Cache-Control` можно добавлять с атрибутом `cacheControlMaxAge` (обратите внимание, что его поведение зависит от атрибута `cacheControlMode`).

Настройка клиентского кеширования посредством ASP.NET MVC

Заголовки `Expires` и `Cache-Control` можно также добавлять программно путем вызова, соответственно, методов `Cache.SetExpires()` и `Cache.SetMaxAge()`.

Ниже приведен пример:

```
// Установить заголовок Cache-Control: max-age в 1 год.  
Response.Cache.SetMaxAge(DateTime.Now.AddYears(1));  
  
// Установить заголовок Expires в 11:00 Р.М.  
// локального времени текущего дня истечения.  
Response.Cache.SetExpires(DateTime.Parse("11:00:00PM"));
```

Заголовки `Cache-Control: max-age` и `Expires` подробно объясняются в разделе “Кеш браузера” главы 12.

Следует отметить, что указание обоих заголовков `Expires` и `Cache-Control` является избыточным – для каждого ресурса должен быть задан только один из них.

Защита кеша

В случае применения заголовка `Expires`, относящегося к далекому будущему, вы должны уведомлять браузер о том, когда кешированный контент изменяется. Если этого не сделать, браузер будет продолжать использовать устаревшую кешированную копию контента.

Поскольку браузер кеширует компоненты по их URL, вы должны изменять URL каким-нибудь образом. Обычно это делается за счет добавления в конец параметра строки запроса, указывающего “версию”. В действительности, средство пакетирования и минимизации ASP.NET MVC предоставляет встроенную функциональную возможность защиты кеша (`cache-busting`), которая позаботится об этом автоматически, когда любой компонент изменится.

Заголовок `Expires`, относящийся к далекому будущему, оказывает влияние на представления страницы только после того, как пользователь снова посетит сайт. Он никак не воздействует на количество HTTP-запросов, когда пользователь заходит на сайт впервые и кеш браузера пуст. Следовательно, влияние на производительность этого улучшения зависит от того, насколько часто пользователи попадают на страницу с наполненным кешем, когда браузер уже содержит все компоненты страницы.

Компоненты, сжатые с помощью GZip

Сжатие текстового контента, такого как HTML, JavaScript, CSS и даже данные JSON, сокращает время, требуемое для передачи компонента по сети, что в конечном итоге значительно улучшает показатель времени отклика.

В среднем сжатие компонентов с помощью GZip приводит к сокращению времени отклика примерно на 70%. Как известно, в более старых браузерах и прокси-серверах имеются проблемы со сжатым контентом, и они могут видеть несоответствие между тем, что ожидалось, и тем, что получено. Однако это крайне случаи, и с прекращением поддержки старых браузеров проблема должна стать несущественной.



Известно, что некоторые прокси-серверы и антивирусные программы удаляют заголовок `Accept-Encoding: gzip, deflate` из HTTP-запросов, приводя к тому, что сервер просто возвращает несжатый контент. Тем не менее, обычно это безопасно. Никаких побочных эффектов кроме снижения производительности не возникает.

Веб-сервер может быть сконфигурирован на сжатие контента на основе типов файлов или MIME-типов либо же он может выяснить это на лету. Хотя определенно имеет смысл включать сжатие для любых текстовых ответов, часто совершенно неэффективно это делать для двоичных компонентов вроде изображений, аудио-роликов и PDF-документов, которые уже являются сжатыми. В таких случаях попытка их дополнительного сжатия может в действительности приводить к *увеличению* размера файла.

Сжатие максимально возможного числа типов файлов – простой способ сокращения общего веса страницы, что обеспечит ее более быструю загрузку, улучшая пользовательское восприятие.

В IIS 7 элемент `<httpCompression>` (<http://www.iis.net/ConfigReference/system.webServer/httpCompression>) задает параметры сжатия HTTP. Сжатие включено по умолчанию при условии, что модуль Performance (Производительность) установлен.

В следующем фрагменте кода представлена стандартная конфигурация этого элемента в файле `ApplicationHost.config` (<http://learn.iis.net/page.aspx/124/introduction-to-applicationhostconfig/>):

```
<httpCompression
    directory="%SystemDrive%\inetpub\temp\IIS Temporary Compressed Files">
    <scheme name="gzip" dll="%Windir%\system32\inetsrv\gzip.dll" />
    <dynamicTypes>
        <add mimeType="text/*" enabled="true" />
        <add mimeType="message/*" enabled="true" />
        <add mimeType="application/javascript" enabled="true" />
        <add mimeType="*/*" enabled="false" />
    </dynamicTypes>
    <staticTypes>
        <add mimeType="text/*" enabled="true" />
        <add mimeType="message/*" enabled="true" />
        <add mimeType="application/javascript" enabled="true" />
        <add mimeType="*/*" enabled="false" />
    </staticTypes>
</httpCompression>
```

Некоторые или даже все эти значения могут быть переопределены за счет указания собственного элемента `<httpCompression>` внутри `<system.webserver>` в файле `web.config` приложения.

Обратите внимание, что динамическое сжатие может увеличить использование центрального процессора, т.к. он будет выполнять сжатие для каждого запроса. Результаты не могут быть эффективно кешированы из-за своей динамической природы.



Если динамический контент является относительно статичным (другими словами, он не изменяется с каждым запросом), его можно по-прежнему кешировать, установив в `true` атрибут `dynamicCompressionBeforeCache` элемента `<urlCompression>` (<http://www.iis.net/ConfigReference/system.webServer/urlCompression>).

Размещайте таблицы стилей в верхней части документа

Помещение таблиц стилей в заголовок документа делает возможной постепенную визуализацию страниц. Поскольку вас заботит производительность, желательно, чтобы страница загружалась постепенно, т.е. браузер должен отображать любой контент, как только это становится возможным. Это особенно важно для страниц с насыщенным контентом и для пользователей с медленными подключениями к Интернету.

В нашем случае HTML-страница является своего рода индикатором хода процесса. Когда браузер загружает страницу постепенно, то заголовок, панель навигации, логотип в верхней части и другие подобные элементы выступают в качестве визуальной подсказки пользователю о том, что страница загружается.

Проблема, связанная с размещением таблиц стилей ближе к нижней части документа, состоит в том, что это приводит к запрету постепенной визуализации во многих браузерах, включая Internet Explorer. Такие браузеры блокируют визуализацию во избежание перерисовки элементов страницы, если их стили изменятся, а это обычно означает, что пользователи вынуждены будут наблюдать пустую страницу.

Размещайте сценарии в нижней части документа

Сценарии блокируют параллельные загрузки. В спецификации HTTP/1.1 (<http://www.w3.org/Protocols/rfc2616/rfc2616-sec8.html#sec8.1.4>) предполагается, что браузеры загружают параллельно не более двух (хотя новые браузеры позволяют немного больше) компонентов на каждое имя хоста. Если изображения обслуживаются несколькими хостами, может потребоваться более двух загрузок в параллельном режиме. Однако пока загружается сценарий, браузер не может начать никаких других загрузок, даже с других хостов (рис. 13.5).

В некоторых ситуациях перенести сценарии в нижнюю часть документа не так-то просто. Например, если в сценарии с помощью `document.write` вставляется часть контента страницы, то такой сценарий нельзя перемещать ниже в рамках страницы. Однако во многих случаях существуют способы обойти такие ситуации. Ниже мы рассмотрим некоторые из этих способов.

Отложенное выполнение сценариев

Синтаксический анализ блока сценариев может быть *отложен* за счет использования в дескрипторе `<script>` атрибута `DEFER`. Атрибут `DEFER` указывает браузерам на то, что они могут продолжать визуализацию.

Тем не менее, различные браузеры обрабатывают этот атрибут по-разному, делая его довольно ненадежным. Важно то, что если выполнение сценария может быть отложено, его также можно переместить в нижнюю часть страницы и добиться более быстрой визуализации этой веб-страницы.

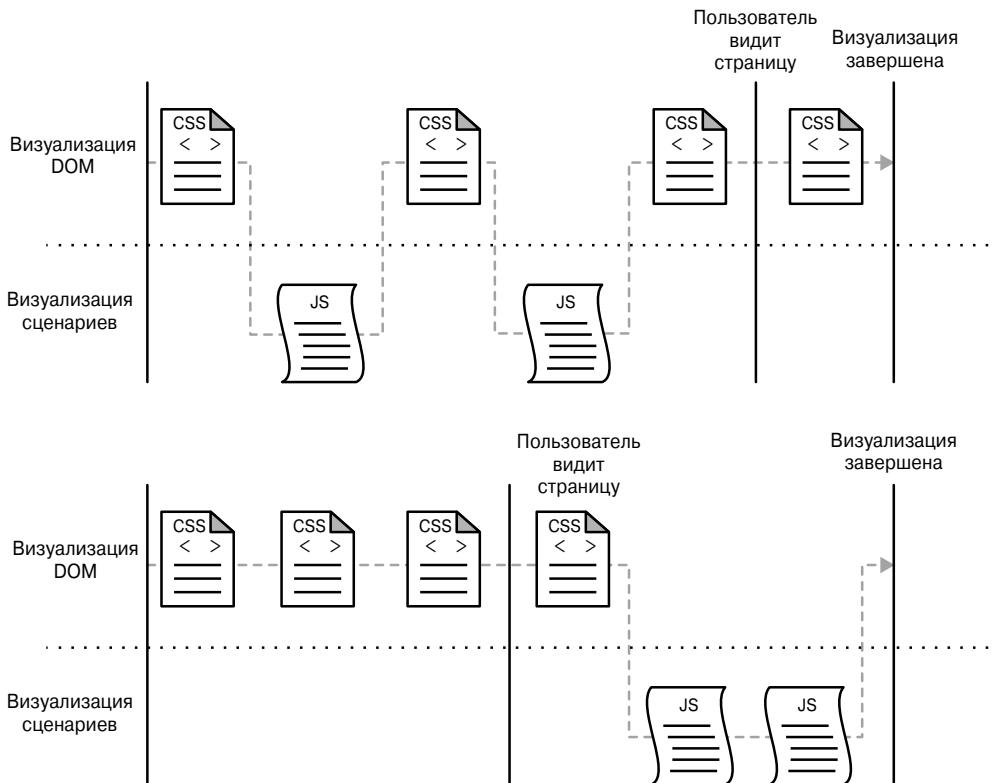


Рис. 13.5. Эффект от визуализации браузером сценариев, расположенных в верхней части документа

Ленивая загрузка сценариев

Некоторые приложения, например, Gmail, используют технологию ленивой загрузки сценариев (<http://googlecode.blogspot.com/2009/09/gmail-for-mobile-html5-series-reducing.html>) для визуализации кода JavaScript внутри блоков комментариев. Браузер просто игнорирует эти комментарии и продолжает визуализировать страницу. Когда блок сценария необходим (в каком-нибудь пользовательском действии), производится доступ к сценарию модуля с отбрасыванием дескрипторов комментария и применением функции `eval()` для синтаксического анализа кода JavaScript. Хотя эта технология не особенно элегантна, она может оказаться более эффективной, чем помещение сценариев в конце страницы или откладывание их выполнения.

Делайте сценарии и стили внешними

Помещение стилей и сценариев в отдельные внешние файлы, как противоположность их встраиванию в HTML-документ, позволяет браузерам кэшировать их. Это ускоряет последующие загрузки страницы, как показано на рис. 13.6.

Включение стилей и сценариев как встроенных делает их некэшируемыми и увеличивает размер ответа. Однако это также сокращает количество HTTP-запросов, поскольку все необходимое загружается как один ресурс.

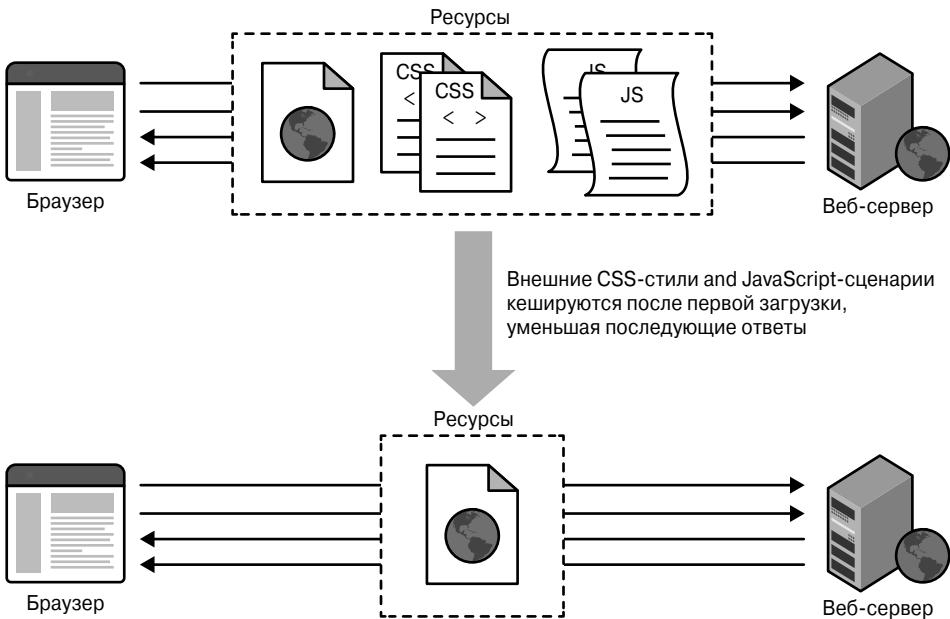


Рис. 13.6. Внешние стили и сценарии

Таким образом, возникает вопрос о том, что выгоднее: сокращение числа запросов или превращение стилей и сценариев во внешние и кешируемые? Ответ на этот вопрос варьируется в зависимости от приложения. Хотя количественное выражение преимуществ часто оказывается сложным, их можно до некоторой степени оценить и измерить. Например, если приложение повторно использует ресурсы на различных страницах, то организация их как внешних обеспечит большие преимущества. Тем не менее, если в приложении имеется лишь несколько страниц или на каждой странице применяются разные ресурсы, то большую выгоду может принести их встраивание (хотя внешние файлы здесь по-прежнему помогут).

Компромисс предусматривает встроенную загрузку ресурсов на посадочной странице и динамическую загрузку внешних ресурсов (используя технологии асинхронной загрузки, такие как AJAX или атрибут `asynchronous` дескриптора `<script>` (<http://davidwalsh.name/html5-asynchron>)). Это ускорит первоначальную загрузку страницы, а последующие просмотры страницы будут извлекать выгоду от работы с кешированными ресурсами. Данный прием реализован на домашней странице Yahoo!.

Сокращайте поиск в DNS

Поиск в DNS – это процесс преобразования имени хоста в его IP-адрес. В среднем он требует 20–120 миллисекунд на запрос, и в течение этого времени браузер не может выполнить никакой другой задачи, поэтому он просто блокируется.

Уменьшение количества HTTP-запросов минимизирует данную проблему, но на странице все равно остается ряд ресурсов (наподобие изображений, таблиц стилей и JavaScript-файлов), которые должны быть запрошены. Сокращение числа уникальных имен хостов на странице позволяет оптимизировать этот процесс, т.к. браузеры обычно кешируют результаты поиска в DNS. Сказанное иллюстрируется на рис. 13.7.



Рис. 13.7. Уменьшение количества поисков в DNS улучшает производительность

Уменьшение количества уникальных имен хостов имеет и побочный эффект: снижение числа параллельных загрузок. Таким образом, должен соблюдаться компромисс между количеством уникальных имен хостов и числом субдоменов для параллельной загрузки. Команда Yahoo! рекомендует разделять такие ресурсы между двумя, тремя или четырьмя хостами для достижения оптимального баланса.

Минимизируйте сценарии и стили

Минимизация – это практический прием удаления необязательных символов из кода для сокращения его размера и тем самым улучшения показателя времени загрузки. Когда код минимизируется, удаляются все комментарии вместе с ненужными пробельными символами (пробелами, символами новой строки и табуляциями).

На рис. 13.8 показан результат минимизации.

```
/* http://meyerweb.com/eric/tools/css/reset/
v2.0 | 2818126
License: none (public domain)
*/
html, body, div, span, applet, object, iframe,
h1, h2, h3, h4, h5, h6, p, blockquote, pre,
a, abbr, acronym, address, big, cite, code,
del, dfn, em, img, ins, kbd, q, samp,
small, strike, strong, sub, sup, tt, var,
b, u, i, center,
dl, dt, dd, ol, ul, li,
fieldset, form, label, legend,
table, caption, tbody, tfoot, thead, tr, th, td,
article, aside, details, embed,
figure, figcaption, footer, header, hgroup,
menu, nav, output, ruby, section, summary,
time, mark, audio, video {
    margin: 0;
    padding: 0;
    border: 0;
    font-size: 100%;
    font: inherit;
    vertical-align: baseline;
}
/* HTML5 display-role reset for older browsers */
article, aside, details, figcaption, figure,
footer, header, hgroup, menu, nav, section {
    display: block;
}
body {
    line-height: 1;
}
ol, ul {
    list-style: none;
}
blockquote, q {
    quotes: none;
}
blockquote:before, blockquote:after,
q:before, q:after {
    content: '';
    content: none;
}
table {
    border-collapse: collapse;
    border-spacing: 0;
}
```

MINIMIZIRUJETSЯ В

```
html,body,div,span,applet,object,
,iframe,h1,h2,h3,h4,h5,h6,p,bloc
kquote,pre,a,abbr,acronym,address
s,big,cite,code,del,dfn,em,img,i
ns,kbd,q,s,samp,small,strike,str
ong,sub,sup,tt,var,b,u,i,center,
dl,dt,dd,ol,ul,li,fieldset,form,
label,legend,table,caption,tbody
,tfoot,thead,tr,th,td,article,as
ide,canvas,details,embed,figure,
figcaption,footer,header,hgroup,
menu,nav,output,ruby,section,sum
mary,time,mark,audio,video(margi
n:0;padding:0;border:0;font-size:100%;font:inherit;vertical-align:baseline)article,aside,det
ails,figcaption,figure,footer,he
ader,hgroup,menu,nav,section(dis
play:block)body(line-height:1)ol,ul(list-
style:none)blockquote,q{quotes:n
one}blockquote:before,blockquote
:after,q:before,q:after{content:'';
content:none}table(border-
collapse:collapse;border-
spacing:0)
```

Рис. 13.8. Минимизация сценариев и стилей уменьшает время отклика

В случае кода JavaScript это снижает время отклика, поскольку уменьшается размер загружаемого файла. Двумя популярными инструментами минимизации кода JavaScript являются JSMin и YUI Compressor. Инструмент YUI Compressor может также минимизировать CSS-стили.

Обфускация – это альтернативная оптимизация, применяемая к исходному коду. В ходе опроса 10 крупнейших веб-сайтов США выяснилось, что с помощью минимизации достигается сокращение размера на 21%, тогда как за счет обфускации – на 25%. Тем не менее, хотя обфускация может давать в результате лучшее сокращение размеров кода, минимизация JavaScript является менее рискованной. Обфускация является более сложной, чем минимизация, поэтому сам шаг обфускации с большей вероятностью может привести к возникновению ошибок.

В дополнение к минимизации внешних сценариев и стилей, встроенные блоки `<script>` и `<style>` могут и должны быть минимизированы. Даже если выполнялось сжатие с помощью GZip сценариев и таблиц стилей, их минимизация по-прежнему сократит размер на 5% и более. С ростом частоты использования и размера JavaScript-сценариев и таблиц стилей будет расти и экономия, достигаемая посредством минимизации кода.

Избегайте перенаправлений

Перенаправление происходит, когда браузер открывает URL, отличающийся от того, что был запрошен. Оно осуществляется с использованием кодов состояния HTTP 301 и 302. Перенаправления считаются дорогостоящими в отношении производительности, т.к. их результаты обычно не кешируются (если только кеширование явно не задано посредством заголовка `Expires` или `Cache-Control`) и они порождают ту же самую задержку, что и новый запрос.

На рис. 13.9 показано, что происходит при перенаправлении.

Запрос				Перенаправление произошло			
Преобразование в IP-адрес	Установление подключения	Первый байт	Загрузка контента	Преобразование в IP-адрес	Установление подключения	Первый байт	Загрузка контента
Перенаправить запрос на тот же самый хост				Перенаправление произошло			
Преобразование в IP-адрес	Установление подключения	Первый байт	Загрузка контента	Преобразование в IP-адрес	Установление подключения	Первый байт	Загрузка контента
Перенаправить запрос на новый хост				Перенаправление произошло			
Преобразование в IP-адрес	Установление подключения	Первый байт	Загрузка контента	Преобразование в IP-адрес	Установление подключения	Первый байт	Загрузка контента

Рис. 13.9. В большинстве случаев перенаправлений следует избегать

Перенаправления могут быть полезны для исправления нарушенных ссылок (например, когда старые страницы были перемещены в новое местоположение), в службах кратких URL или для переадресации пользователей из множества (в чем-то похожих) доменных имен на единственный домен (скажем, из `wikipedia.net` на `wikipedia.com`).

Хотя приведенные выше случаи являются допустимыми применениями перенаправления, которых нельзя избежать, довольно часто перенаправления происходят без ведома разработчика. К примеру, в приложении ASP.NET MVC (или любом приложении, выполняющемся на сервере IIS) обращение к `http://www.ebuy.biz/Home/About`

приведет к перенаправлению на `http://ebuy.biz/Home/About/` – т.е. к тому же самому действию, но с завершающим символом `/`. Это можно исправить, сделав перенаправление “постоянным” (код состояния HTTP 301).

Ниже приведен пример HTTP-заголовков в ответе 301:

```
HTTP/1.1 301 Moved Permanently
Location: http://yourhostname.com/Home/About
Content-Type: text/html
```

Существует много способов для обработки этого, например:

- написание собственного HTTP-модуля;
- выполнение перенаправления на контроллере;
- использование модуля IIS URL Rewrite (<http://www.iis.net/download/urlrewrite>) для перезаписи URL.

Модуль URL Rewrite – это HTTP-модуль, встроенный непосредственно в IIS. Конфигурировать модуль URL Rewrite можно с помощью элемента `system.webServer⇒rewrite` в файле `web.config`.

В следующем фрагменте кода показано, как выполнить такое перенаправление:

```
<rewrite>
  <rules>
    <!-- Удалить завершающую косую черту из URL -->
    <rule name="Strip trailing slash" stopProcessing="true">
      <match url=".*/$" />
      <conditions>
        <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="true" />
        <add input="{REQUEST_FILENAME}"
              matchType="IsDirectory" negate="true" />
      </conditions>
      <action type="Redirect" redirectType="Permanent" url="{R:1}" />
    </rule>
  </rules>
</rewrite>
```

В приведенном фрагменте добавляются правила, в которых задействованы регулярные выражения для обнаружения, содержит ли запрос завершающую косую черту. Если это так, завершающая косая черта удаляется, и обратно возвращается заголовок ответа постоянного перенаправления с перезаписанным URL.

Удаляйте дублированные сценарии

Повторное включение одного и того же JavaScript-файла на странице наносит урон производительности – и это не столь уж необычная ситуация, как можно было бы подумать. Обзор 10 крупнейший веб-сайтов США показал, что два из них содержат дублированные сценарии. Вероятность дублирования сценариев на одной и той же странице повышают два фактора: размер команды разработчиков и количество сценариев. Когда дублирование случается, оно может привести к снижению производительности.

Дублированные сценарии в результате порождают ненужные HTTP-запросы в Internet Explorer (но не в Firefox). В Internet Explorer двукратное включение внешнего сценария, не являющегося кешируемым, вызывает генерацию двух HTTP-запросов во время загрузки страницы. Даже если сценарий кешируемый, при перезагрузке страницы пользователем произойдут дополнительные HTTP-запросы.

Кроме генерации излишних HTTP-запросов, непроизводительно расходуется время на многократную оценку сценария. Это избыточное выполнение сценария JavaScript происходит как в Firefox, так и в Internet Explorer, и не зависит от того, является сценарий кешируемым или нет.

Один из способов избежать случайного включения одного и того же сценария дважды заключается в реализации модуля управления сценариями внутри используемой системы шаблонов. Типичный метод включения сценария предусматривает применение дескриптора `<script>` на HTML-странице.

Конфигурируйте теги ETag

Тег сущности (Entity Tag – ETag) представляет собой строку, которая уникальным образом идентифицирует специфичную версию ресурса или компонента, такого как изображение, таблица стилей или сценарий. Теги ETag – это механизм, который веб-серверы и браузеры используют для определения, соответствует ли компонент в кеше браузера компоненту на сервере-источнике.

Теги ETag были добавлены, чтобы предоставить механизм для проверки достоверности сущностей, являющихся более гибкими и надежными, чем дата последнего изменения, которая обеспечивает только слабую форму такой проверки (браузеры применяют эвристические алгоритмы для выяснения, извлекать ли ресурс из сервера, и каждый браузер применяет эти алгоритмы по-разному).

Ниже показано, как выглядит заголовок ETag ответа:

```
HTTP/1.1 200 OK
Last-Modified: Tue, 29 May 2012 00:00:00 GMT
ETag: "8e12af-3bd-632a2d18"
Content-Length: 14625
```

Для проверки достоверности ресурса на более поздней стадии браузер использует заголовок `If-None-Match`, передавая ETag обратно серверу-источнику. Сервер возвращает код состояния 304, если теги Etag совпадают (в этом случае сокращая размер ответа на 14 625 байтов):

```
GET /images/logo.png HTTP/1.1
Host: yourhostname.com
If-Modified-Since: Tue, 29 May 2012 00:00:00 GMT
If-None-Match: "8e12af-3bd-632a2d18"
HTTP/1.1 304 Not Modified
```

Если приложение работает с веб-фермой или кластером веб-серверов, важно, чтобы каждый сервер назначал ресурсу один и тот же ETag – иначе браузер будет считать их как отличающиеся версии и загружать полный ресурс при обслуживании разными серверами, сводя на нет само предназначение ETag.

К сожалению, веб-серверы Apache и IIS включают в ETag такие данные, которые делают практически невозможной генерацию одинаковых тегов ETag на серверах внутри веб-фермы. Это значительно снижает шансы на успешное прохождение проверки достоверности в браузере при сценарии с веб-фермой. В таких случаях для выполнения этой проверки может быть проще применять заголовок `Last-Modified`.

Если вы изберете данный путь, то должны удалить заголовок ETag из ответа, что сократит размеры запроса и ответа HTTP. В статье Microsoft Support, доступной по адресу <http://support.microsoft.com/?id=922733>, рассказывается, каким образом удалить теги ETag из IIS.

Обратите внимание, что необходимо устанавливать либо заголовок Last-Modified, либо ETag. Установка обоих является избыточной.

Измерение производительности клиентской стороны

Чтобы “оптимизировать” нечто, сначала нужно иметь возможность его “измерить” или “определить количество”. Без профилирования либо инструментов для измерений невозможно идентифицировать узкие места, и уж тем более обосновать улучшения.

Для этих целей доступно множество инструментальных средств, простейшим из которых, пожалуй, является YSlow (<http://yslow.org>). Инструмент YSlow работает со многими браузерами, и хотя далее в этой главе внимание сосредоточено на Firefox, можно использовать любой желаемый браузер, поскольку ключевые концепции и приемы остаются неизменными. Инструмент YSlow имеет дело с 23 из 35 правил, которые команда производительности Yahoo! представила как измеряемые количественно и проверяемые.

Первым делом понадобится установить дополнение YSlow для Firefox (<https://addons.mozilla.org/en-US/firefox/addon/5369>). После этого можно создать новый базовый проект ASP.NET MVC и посмотреть, как он оценивается по эталонам YSlow.

Выберите пункт меню File⇒New⇒Project (Файл⇒Создать⇒Проект), укажите тип проекта ASP.NET MVC 4 Web Application (Веб-приложение ASP.NET MVC 4) и затем шаблон Internet Application (Интернет-приложение), как показано на рис. 13.10.

Не изменения ни единой строчки кода, скомпилируйте и запустите приложение. Если браузером по умолчанию является не Firefox, откройте Firefox и перейдите на новое приложение.

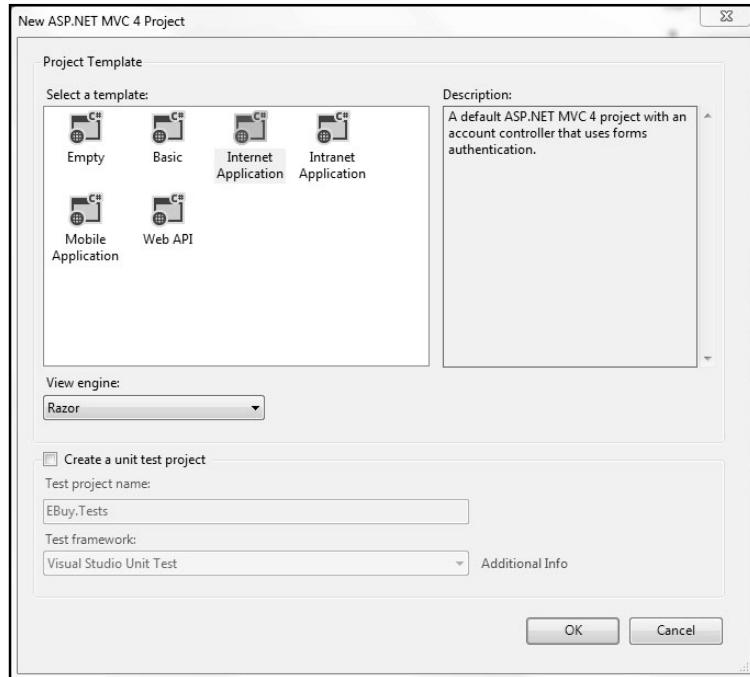


Рис. 13.10. Создание базового приложения ASP.NET MVC 4

Активизируйте Firebug, перейдите на вкладку YSlow (рис. 13.11) и щелкните на кнопке Run Test (Запустить тест).

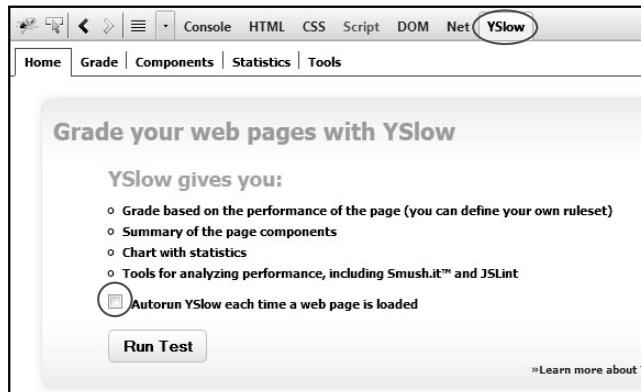


Рис. 13.11. Вкладка YSlow в Firebug

При использовании стандартного набора правил (YSlow (V2)) результаты впечатляют. Стартовый шаблон уже следует многим рекомендациям и, как видно на рис. 13.12, получает оценку A почти во всех разделах. В целом страница имеет оценку B.

A screenshot of the Grade report for the Home Page in Firebug. The title bar says 'Home Page. Welcome to EBuy!'. The tab bar shows 'Console', 'HTML', 'CSS', 'Script', 'DOM', 'Net', and 'YSlow'. Below the tab bar is a navigation menu with links 'Home', 'Grade', 'Components', 'Statistics', and 'Tools'. The main content area shows a grade of 'B' with the message 'Overall performance score 81 Ruleset applied: YSlow(V2) URL: http://localhost:24796/'. It lists 'ALL (22)' items under 'FILTER BY: CONTENT (6) | COOKIE (2) | CSS (6) | IMAGES (2) | JAVASCRIPT (4) | SERVER (6)'. A large section titled 'E Make fewer HTTP requests' contains a list of items: 'F Use a Content Delivery Network (CDN)', 'A Avoid empty src or href', 'F Add Expires headers', 'A Compress components with gzip', 'A Put CSS at top', 'A Put JavaScript at bottom', 'A Avoid CSS expressions', 'n/a Make JavaScript and CSS external', 'A Reduce DNS lookups', 'A Minify JavaScript and CSS', 'A Avoid URL redirects', 'A Remove duplicate JavaScript and CSS', 'F Configure entity tags (ETags)', 'A Make AJAX cacheable', 'A Use GET for AJAX requests', 'A Reduce the number of DOH elements', 'A Avoid HTTP 404 (Not Found) error', 'A Reduce cookie size', 'A Use cookie-free domains', 'A Avoid AlphaImageLoader filter', 'A Do not scale images in HTML', and 'A Make favicon small and cacheable'. To the right of this list is a detailed explanation of the 'Grade E on Make fewer HTTP requests' rule, mentioning that the page has 13 external stylesheets and suggesting to combine them into one. There is also a link to 'Read More'.

Рис. 13.12. Стартовый шаблон имеет довольно высокую общую оценку

Первая низкая оценка встречается для правила #1, Make fewer HTTP requests (Делайте меньше HTTP-запросов). На вкладке Grade (Оценка) предлагается незамедлительный способ решения этой проблемы – комбинирование друг с другом 13 файлов с таблицами стилей. Перейдя на вкладку Components (Компоненты), показанную на рис. 13.13, вы увидите, сколько компонентов различных типов запрашивает страница. Обратите внимание, что в стартовом шаблоне используются только два файла JavaScript (JS), но по мере роста приложения будут появляться дополнительные JavaScript-файлы, которые нужно будет пакетировать вместе.

† TYPE	SIZE (KB)	GZIP (KB)	COOKIE RECEIVED (bytes)
doc (1)	5.2K		
is (2)	283.3K		
css (13)	53.6K		
cssimage (6)	3.7K		
favicon (1)	32.0K		

* type column indicates the component is loaded after window.onload event
† denotes 1x1 pixels image that may be image beacon

Copyright © 2012 Yahoo! Inc. All rights reserved.

Рис. 13.13. Вкладка Components показывает, сколько компонентов запрашивает эта страница

Давайте посмотрим, как можно воспользоваться встроенным средством пакетирования ASP.NET MVC 4 для сокращения количества HTTP-запросов на этой странице.

Ввод в работу ASP.NET MVC

Инфраструктура ASP.NET MVC 4 и платформа .NET Framework 4.5 предоставляют новую библиотеку System.Web.Optimization, которая предлагает готовую поддержку пакетирования и минимизации. Она обеспечивает базовые возможности для пакетирования различных ресурсов в соответствие со специальными правилами (как будет показано в последующих разделах), а также встроенный инструмент минимизации JavaScript-кода и таблиц стилей. Эта библиотека также включает автоматическую защиту кеша, объявляющую недействительным кеш браузера при изменении любого контента. Другим важным средством, которое она предлагает, является возможность автоматического дублирования таблиц стилей и файлов сценариев (до тех пор, пока они не находятся в разных путях).

Для большинства сценариев этого достаточно. Тем не менее, если приложению требуется большее, всегда можно выбрать какое-нибудь решение от независимых разработчиков.

Пакетирование и минимизация

Заглянув в дескриптор `head` внутри файла `Layout.cshtml`, вы заметите два новых вспомогательных класса, предоставляемых библиотекой `System.Web.Optimization` – `@Styles` и `@Scripts`:

```
@Styles.Render("~/Content/themes/base/css", "~/Content/css")
@Scripts.Render("~/bundles/modernizr")
```

Эти классы пакетируют и минимизируют, соответственно, таблицы стилей и файлы JavaScript или ресурсы.

Метод `Render()` принимает список виртуальных путей для визуализации. Во время выполнения эти пути транслируются в обычные HTML-дескрипторы, примерно так:

```
<link href="/Content/site.css" rel="stylesheet" type="text/css" />
<script src="/Scripts/modernizr-2.0.6.js" type="text/javascript"></script>
```

Как будет показано в следующем разделе, вы обладаете полным контролем над определением этих пакетов. Например, как было проиллюстрировано выше в примере, можно создать множество пакетов и визуализировать их в единственном вызове. Здесь в одном вызове визуализируются два пакета стилей – один для базовой темы и один для стандартной таблицы стилей сайта.

Просматривая дальше, вы увидите, что чуть выше закрывающего дескриптора `body` находится оставшийся JavaScript-вызов `Render()`:

```
@Scripts.Render("~/bundles/jquery")
```

Следовательно, стартовый шаблон автоматически придерживается упомянутых ранее правил “размещать таблицы стилей в верхней части” и “размещать JavaScript-сценарии в нижней части”.



Единственным исключением является файл `modernizr.js`, который визуализируется в заголовке документа. Роль JavaScript-библиотеки Modernizr заключается в определении функциональных возможностей (таких как видео, аудио, SVG (масштабируемая векторная графика), новейшие средства HTML 5 и т.д.), которые поддерживаются целевым браузером, и в присоединении имен классов таблиц стилей к дескриптору `<head>`. Используя эти классы, страницу можно стилизовать и снабжать сценариями, чтобы изящно обходить отсутствие отдельной функциональной возможности, от которой зависит страница. Перемещение Modernizr в конец страницы отложит это определение вплоть до момента загрузки страницы, поэтому в самом начале стили работать не будут, чтобы не создавать искажения во время загрузки страницы (некоторые элементы могут выглядеть запорченными, но после загрузки страницы станут отображаться корректно).

Определение пакетов

Для определения пакета необходимо вызвать метод `BundleCollection.Add()` (в `System.Web.Optimization`) и передать ему экземпляр `ScriptBundle` или `StyleBundle`. Экземпляр `ScriptBundle` создается за счет указания виртуального пути (который будет использоваться в представлениях) и включения одного и более сценариев, как показано ниже:

```
// Групповое включение – включить все сценарии, которые начинаются с "jquery-1".
var jQueryBundle = new ScriptBundle("~/bundles/jquery").Include(
    "~/Scripts/jquery-1.*");
```

```
// Явное включение файла сценариев.  
var jQueryValBundle = new ScriptBundle("~/bundles/jqueryval").Include(  
    "~/Scripts/jquery.unobtrusive-ajax.js",  
    "~/Scripts/jquery.validate.js",  
    "~/Scripts/jquery.validate.unobtrusive.js")
```

Экземпляр StyleBundle создается аналогично:

```
var siteBundle = new StyleBundle("~/Content/css").Include("~/Content/site.css")
```

Стартовый шаблон делает это автоматически для включенных ресурсов — загляните в App_Start\BundleConfig.cs. Этот файл преднамеренно помещен в папку App_Start, подчеркивая тот факт, что данный код должен выполняться только один раз, во время начальной загрузки приложения. Тем не менее, определять пакеты можно в любых местах проекта при условии, что вы не забудете их зарегистрировать во время фазы начальной загрузки приложения:

```
public class MvcApplication : System.Web.HttpApplication  
{  
    protected void Application_Start()  
    {  
        AreaRegistration.RegisterAllAreas();  
        FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);  
        RouteConfig.RegisterRoutes(RouteTable.Routes);  
        // Зарегистрировать пакеты.  
        BundleConfig.RegisterBundles(BundleTable.Bundles);  
    }  
}
```

Включение пакетов

Вспомните, что в первом тесте инструмент YSlow сообщил о слишком большом количестве HTTP-запросов. В результате этого возникает вопрос: если весь вспомогательный код уже на месте, то почему пакетирование не скомбинировало их с целью уменьшения числа запросов?

Причина в том, что в режиме отладки (Debug) пакетирование и другие оптимизации автоматически отключены, чтобы сделать более удобными разработку и отладку кода. Принудительно включить эти оптимизации в режиме отладки можно, установив BundleTable.EnableOptimizations в true.

Чтобы увидеть это в действии, скомпилируйте приложение в режиме выпуска (Release) и запустите повторно тест YSlow. Общей оценкой по-прежнему остается *B* (из-за влияния других факторов), но YSlow теперь отмечает высокой оценкой сокращение количества HTTP-запросов (рис. 13.14). На вкладке Components (рис. 13.15) отражено пакетирование в действии и присутствуют только два файла с таблицами стилей, как и было определено ранее: /Content/themes/base/css и /Content/css.

Если просмотреть исходный код для URL таблиц стилей, также можно отметить выполненную минимизацию.

Защита кеша

Вспомните, что браузеры кешируют ресурсы на основе URL. Всякий раз, когда страница запрашивает ресурс, браузер сначала проверяет в своем кеше наличие ресурса с подходящим URL. Если он присутствует, то вместо извлечения нового ресурса из сервера просто используется кешированная копия. Таким образом, чтобы иметь возможность работы с кешем браузера, URL не должен изменяться между посещениями.

Grade B Overall performance score 87 Ruleset applied: YSlow(V2) URL: http://localhost:24796/

ALL (23) FILTER BY: [CONTENT \(6\)](#) | [COOKIE \(2\)](#) | [CSS \(6\)](#) | [IMAGES \(2\)](#) | [JAVASCRIPT \(4\)](#) | [SERVER \(6\)](#)

A Make fewer HTTP requests

F Use a Content Delivery Network (CDN)	Grade A on Make fewer HTTP requests
A Avoid empty src or href	

Рис. 13.14. Использование пакетов для сокращения числа HTTP-запросов

Components The page has a total of **12** components and a total weight of **91.2K bytes**

TYPE	SIZE (KB)	GZIP (KB)	COOKIE RECEIVED (bytes)	COOKIE SENT (bytes)	HEADERS
[+] doc (1)	4.3K				
[+] js (2)	100.5K				
[+] css (2)	33.1K				
css	24.9K	5.6K			🔍 http://localhost:24796/
css	8.1K	2.7K			🔍 http://localhost:24796/
[+] cssimage (6)	3.7K				
[+] favicon (1)	32.0K				

Рис. 13.15. Таблицы стилей теперь пакетированы в два файла

Однако при этом веб-разработчику придется разрешить одну дилемму. Когда изменяется код в файлах JavaScript или таблиц стилей, необходимо, чтобы браузер получал обновленную копию, а не кешированную. Сохранение URL неизменным усложняет дело, т.к. браузер продолжит пользоваться старой кешированной копией. Распространенное решение предусматривает добавление к URL номера версии, например:

```
<link type="text/css" rel="stylesheet" href="/Content/site.css?v=1.0">
```

Теперь в случае изменения контента можно просто увеличить номер версии и браузер загрузит новую копию. Однако метод ручного редактирования номеров версий довольно утомителен и предрасположен к ошибкам.

Пакетирование автоматически заботится об этой проблеме, добавляя хеш-код пакета в виде параметра запроса, как показано ниже:

```
<link type="text/css" rel="stylesheet"
      href="/Content/themes/base/css?v=UM624qf1uFt8dYtiIV9PCmYhsyeewBIwY4Ob0i80dW81">
```

Теперь любое изменение, внесенное в файл, автоматически приводит к генерации нового хеш-кода и внедрению его на страницу. Браузер, столкнувшись с отличающимся URL, будет извлекать более новую копию вместо использования кэшированной версии.

Резюме

В этой главе были представлены базовые правила, позволяющие обеспечить более быстрое выполнение страниц. Вы узнали, как внедрять некоторые из этих правил в IIS, и пользоваться такими встроенными средствами ASP.NET MVC, как пакетирование, минимизация и защита кеша. Приведенные в главе советы довольно просты в реализации, и они обеспечат изначальные преимущества в оптимизации производительности разрабатываемых страниц.

Расширенная маршрутизация

В первой главе этой книги было дано краткое введение в фундаментальные основы маршрутизации ASP.NET MVC. В настоящей главе будет описан стандартный маршрут, который Visual Studio генерирует при создании нового проекта ASP.NET MVC, и показано, как инфраструктура маршрутизации использует этот маршрут для определения контроллера и действия, которые должны применяться для выполнения каждого запроса.

По большей части, вам не придется беспокоиться ни о чем более сложном, чем стандартный маршрут, изначально генерируемый Visual Studio. Этот стандартный маршрут следует принятому в ASP.NET MVC соглашению и позволяет создавать новые контроллеры и действия, не особенно заботясь о том, каким образом механизм маршрутизации будет находить их. Более глубокое понимание мощной инфраструктуры маршрутизации ASP.NET MVC понадобится в ситуации, когда приложение выходит за рамки таких общих сценариев, и требуется оперирование за пределами стандартного шаблона URL и значений по умолчанию.

В этой главе мы оставим в стороне обычный маршрут ASP.NET MVC и подробно рассмотрим мощный механизм маршрутизации, который управляет приложениями ASP.NET MVC. Мы начнем с обсуждения о том, почему URL настолько важны для опыта взаимодействия с веб-приложением, и как URL приложения могут влиять на позицию сайта в результатах выдачи поисковых систем. Затем мы расскажем о создании более сложных маршрутов, исследуя различные шаблоны URL. Также будет рассмотрен вопрос определения ограничений на маршрутах и полезный инструмент под названием *Glimps*, который помогает отлаживать отказы при маршрутизации. Наконец, мы продемонстрируем способы расширения базовой инфраструктуры маршрутизации, обеспечивающие улучшение опыта взаимодействия, и посмотрим, как воспользоваться расширяемостью механизма маршрутизации для создания собственной логики маршрутизации.

Нахождение пути

Понятие *нахождение пути* (*wayfinding*) относится ко всем способам, которыми люди ориентируются в пространстве и перемещаются с места на место. В контексте просмотра World Wide Web одним из наиболее естественных механизмов нахождения пути являются URL. Фактически именно так большинство людей изначально ориентируются в World Wide Web: открывают браузер и вводят какой-нибудь URL.

Например, практически всем известен URL поисковой системы Yahoo!:

www.yahoo.com

Адреса URL представляют собой простой и интуитивно понятный способ для пользователей попасть туда, куда они хотят, и что более важно – получить информацию, которую они ищут. URL просты для запоминания и передачи другим людям.

На заре World Wide Web простота была присуща как доменному имени, так и в целом URL, как можно видеть в следующем URL для специфической страницы из версии Yahoo!, существовавшей в 1996 г.:

http://www.yahoo.com/Computers_and_Internet/Software/Data_Formats/HTML/HTML_2_0/

В приведенном примере URL структура каталогов вполне очевидна и легко читааема. По этой же причине такой URL иногда называют “уязвимым к атакам”. Под этим имеется в виду то, что пользователь может легко изменять части URL для перехода на более высокие уровни внутри иерархии сайта, а также делать хорошо обоснованные предположения о том, какие другие категории URL могут быть определены на данном сайте.

Простые URL являются важным – и часто упускаемым из виду – аспектом опыта взаимодействия, который веб-сайт обеспечивает своим пользователям. Свыше 10 лет тому назад известный эксперт по удобству использования Якоб Нильсен отметил, что хорошая структура URL привносит свой вклад в удобство использования веб-сайта. Затем проведенные в 2007 г. командой Нильсена, Эдвардом Катреллом и Чживэй Гуань из Microsoft Research отдельные исследования по отслеживанию глаз показали, что пользователи тратят 24% времени своего взгляда на просмотр URL.

По мере роста популярности инфраструктур для построения веб-страниц, разработчики осознали, что URL можно применять не только в качестве средства для доступа к ресурсам на сайтах, но также как место для хранения информации о действии пользователя и состоянии приложения. Поскольку разработчики использовали этот прием чаще и чаще, URL стали в большей степени ориентироваться на машины, на которых выполнялись веб-сайты, а не на пользователей, работающих с ними, и начали наполняться непонятными кодами и различными идентификаторами баз данных, например:

<http://demo.com/store.aspx?v=c&p=56&id=1232123&s=12321-12321321312-12312&s=0&f=red>

Такие виды URL однозначно стало сложно читать и запоминать, не говоря уже о том, чтобы диктовать их, скажем, по телефону.

К счастью, те же самые веб-разработчики впоследствии начали понимать то, что Нильсену и другим было давно известно: структура URL важна, и URL играют немаловажную роль в удобстве использования веб-сайта. Чтобы решить возникшие проблемы и вернуть осмысленность URL, новейшие инфраструктуры для разработки веб-приложений вводят концепцию *маршрутизации*, представляющую собой уровень косвенности между URL и функциональностью приложения, к которому они относятся.

Маршрутизация позволяет разместить фасад URL над логикой приложения. Например, применение маршрутизации к показанному выше URL даст возможность скрыть конечную точку *store.aspx* и все значения строки запроса, взамен отобразив дружественные сегменты URL на параметры, которые нужны для функционирования страницы.

Другими словами, неуклюжий URL можно преобразовать во что-то более осмысленное:

<http://example.com/store/toys/RadioFlyer/ClassicRedWagon/12321-12321321312-12312/red>

Хотя этот URL передает в основном ту же самую информацию, что и предшествующий, он делает это намного более дружественным к пользователю способом. Здесь легко можно сказать, что мы находимся в магазине и покупаем товар “Radio Flyer Classic Red Wagon” в категории “toys”. Пользователь может даже удалить часть URL, чтобы поэкспериментировать с навигацией на другие страницы сайта.

В мире ASP.NET MVC вместо функционирования в качестве фасада для конечных точек .aspx, механизм маршрутизации выступает в роли фасада для контроллеров и действий приложения. Таким образом, поскольку вы строите приложение ASP.NET MVC, полезно продумать URL, которые приложение будет открывать внешнему миру.

URL И ПОИСКОВАЯ ОПТИМИЗАЦИЯ

В дополнение к общему удобству использования веб-сайта, создание более дружественных URL может дать сайту еще одно очень важное преимущество: улучшенный рейтинг в поисковых системах. Более дружественные в плане человеческого восприятия URL также являются более дружественными в отношении поисковых систем. Оптимизация URL – это часть более крупной технологии улучшения рейтинга в поисковых системах, которая называется *поисковой оптимизацией* (Search Engine Optimization – SEO).

Цель поисковой оптимизации заключается в приведении сайта в такое состояние, при котором увеличиваются ранги страниц сайта в результатах выдачи поисковых систем. Поскольку поисковый рейтинг является важным фактором для веб-сайта, существует несколько советов по поисковой оптимизации, которые могут повлиять на проектирование URL приложения. Однако имейте в виду, что в поисковой оптимизации остается немного “черной магии”, и каждая поисковая система применяет собственные патентованные – и обычно засекреченные – алгоритмы для назначения рейтинга веб-страницам, поэтому четкие правила отсутствуют.

Тем не менее, в целом имеется несколько хороших советов, которые следует принимать во внимание при оптимизации сайта.

- **Лучше использовать короткие URL.** В Google заявляют, что их алгоритмы назначают меньший вес словам, встречающимся после первых пяти слов в URL, так что более длинные URL в действительности не дают каких-то преимуществ. Вдобавок короткие URL помогают повысить удобство использования и читабельность.
- **Разделять множество слов с помощью дефисов, а не подчеркиваний.** Google будет интерпретировать дефисы как разделители слов и индексировать каждое слово по отдельности.
- **Придерживаться нижнего регистра символов.** Осознанно применяйте прописные буквы в URL и старайтесь придерживаться нижнего регистра символов, если это возможно. Большинство поисковых систем следуют стандарту HTTP, который утверждает, что URL являются чувствительными к регистру символов; по этой причине поисковая система рассматривает Page1.htm и page1.htm как две разных страницы. Это может привести к двукратной индексации контента и в результате повлечь за собой штраф в отношении рейтинга указанных страниц.



За дополнительными сведениями об оптимизации URL обратитесь к статье по адресу <http://www.seomoz.org/blog/seo-cheat-sheet-anatomy-of-a-url>.

Построение маршрутов

Надеемся, что теперь вы хорошо осознаете важность URL в формировании общего опыта взаимодействия с вашим сайтом. Имея это в виду, давайте посмотрим, как использовать инфраструктуру маршрутизации ASP.NET MVC для управления тем, какие URL пользователи будут применять для доступа к сайту. Мы начнем с более подробного анализа различных способов определения шаблона URL в маршруте. Мы будем использовать URL, определенный в стандартном маршруте, который был построен при создании нового проекта ASP.NET MVC в качестве опорной точки:

```
routes.MapRoute(  
    "Default", // Имя маршрута  
    "{controller}/{action}/{id}", // URL с параметрами  
    new { controller = "Home", action = "Index", id = UrlParameter.Optional }  
) ;
```

Строка URL в маршруте формируется за счет комбинации последовательности из одного или более “сегментов”. Сегменты могут быть константами или заполнителями и разделяются символом косой черты.



Поскольку маршруты всегда являются относительными корня приложения, они не могут начинаться с символа косой черты (/) или тильды (~). Механизм маршрутизации будет генерировать исключение, если сталкивается с маршрутом, нарушающим это правило.

Каждый из этих сегментов можно назвать заполнителем, т.к. он заключен в фигурные скобки:

```
{controller}
```

В этом случае каждый заполнитель является также и одиночным сегментом, разделенным с помощью косой черты. Допускается создавать сегменты, которые содержат множество заполнителей, отделяя каждый заполнитель строковой константой, как показано ниже:

```
{param1}-{param2}-{param3}
```

Когда механизм маршрутизации разбирает URL, он извлекает значения в каждой позиции заполнителя и применяет их для наполнения экземпляра класса `RouteData`. Этот класс поддерживает словарь всех важных значений, содержащихся в маршруте, и используется самой инфраструктурой ASP.NET MVC Framework. Также важно знать, что значения, добавляемые в этот словарь, по умолчанию преобразуются в строки.

В случае стандартного маршрута, показанного ранее, это означает, что если действительный URL запроса выглядит следующим образом:

```
http://demo.com/Home/Index/1234
```

то механизм маршрутизации разберет три пары “ключ/значение”, приведенные в табл. 4.1, в экземпляр класса `RouteData`.

Таблица 14.1. Пары “ключ/значение”, передаваемые экземпляру класса `RouteData`

Параметр	Значение
{controller}	Home
{action}	Index
{id}	1234

Стандартные и необязательные параметры маршрута

Как было показано в главе 1, система маршрутизации использует значения {controller} и {action} для выяснения того, экземпляр какого контроллера создавать и какое действие запускать. Однако вспомните, что URL, загруженный в браузер, не содержит никаких значений для заполнителей URL. Именно здесь в игру вступает третий параметр метода MapRoute(), позволяя устанавливать стандартные значения для любого заполнителя.

Если вернуться и еще раз взглянуть на стандартный маршрут, можно заметить, что в вызове MapRoute() создается анонимный тип (специальный объектный тип, который компилятор генерирует на лету), устанавливающий стандартные значения для каждого из трех определенных заполнителей:

```
routes.MapRoute(
    "Default",                                     // Имя маршрута
    "{controller}/{action}/{id}",                  // URL с параметрами
    new { controller = "Home", action = "Index", id = UrlParameter.Optional }
);
```

Теперь должно быть понятно, каким образом приложение знает, что по умолчанию нужно загружать HomeController. Несмотря на то что URL запроса не содержит значения для контроллера или действия, маршрут имеет заданные стандартные значения, которые система маршрутизации применяет для нахождения стандартного контроллера и действия. При оценке механизмом маршрутизации URL стандартные значения вставляются в словарь RouteData, и если значение заполнителя обнаруживается во время разбора URL, оно просто перезаписывает стандартное значение.

Давайте рассмотрим ряд других примеров URL маршрутов. В следующем примере в начале URL находится константный сегмент. Это означает, что для соответствия этому маршруту URL запроса должен содержать этот сегмент:

```
routes.MapRoute(
    "Default",
    "Admin/{controller}/{action}/{id}",
    new { controller = "Home", action = "Index", id = UrlParameter.Optional }
);
```

Ниже перечислены некоторые примеры URL, соответствующие данному маршруту:

- <http://demo.com/Admin>
- <http://demo.com/Admin/Home>
- <http://demo.com/Admin/Home/Index>
- <http://demo.com/Admin/Home/Index/1234>

```
routes.MapRoute(
    "Default",
    "{site}/{controller}/{action}/{id}",
    new { controller = "Home", action = "Index", id = UrlParameter.Optional }
);
```

Вот несколько примеров URL, которые удовлетворяют этому маршруту:

- <http://demo.com/Admin>
- <http://demo.com/Store/Home>
- <http://demo.com/Store/Home/Index>
- <http://demo.com/Store/Home/Index/1234>

В приведенном ниже определении маршрута содержится единственный заполнитель для идентификатора, а контроллер и действие получают стандартные значения:

```
routes.MapRoute(
    "Default",
    "{id}",
    new { controller = "Home", action = "Index", id = UrlParameter.Optional }
);
```

Этот маршрут дает совпадения для URL следующего вида:

- <http://demo.com/>
- <http://demo.com/1234>

В показанном далее маршруте комбинируется константный сегмент с единственным заполнителем, при этом контроллер и действие снова получают стандартные значения:

```
routes.MapRoute(
    "Default",
    "users/{id}",
    new { controller = "Home", action = "Index", id = UrlParameter.Optional }
);
```

Ниже перечислены примеры URL, которые будут соответствовать данному маршруту:

- <http://demo.com/users>
- <http://demo.com/users/1234>

```
routes.MapRoute(
    "Default",
    "category/{id}/export{type}.{format}/",
    new { controller = "Category", action = "Export" }
);
```

А вот пример URL, соответствующего этому маршруту:

- <http://demo.com/category/123abc/exportEvents.json>

Порядок и приоритет маршрутизации

Как только приложение становится более сложным, наверняка понадобится регистрировать несколько маршрутов. При этом важно обдумать порядок, в котором они будут регистрироваться. Когда механизм маршрутизации пытается найти совпадающий маршрут, он просто проходит по коллекции маршрутов и останавливается, как только обнаруживает совпадение.

Такое поведение может вызвать множество проблем, если не ожидать его. Взглядите на следующий фрагмент, в котором регистрируются два маршрута:

```
routes.MapRoute(
    "generic",
    "{site}",
    new { controller = "SiteBuilder", action = "Index" }
);
routes.MapRoute(
    "admin",
    "Admin",
    new { controller = "Admin", action = "Index" }
);
```

Первый маршрут содержит единственный сегмент заполнителя и устанавливает параметр controller в стандартное значение SiteBuilder. Второй маршрут содержит единственный константный сегмент и устанавливает параметр controller в стандартное значение Admin.

Оба маршрута совершенно допустимы, но порядок, в котором они отображаются, может привести к появлению неожиданных проблем, потому что первый маршрут соответствует практически любому введенному значению. Это означает, что для `http://demo.com/Admin` на нем произойдет первое совпадение, и поскольку механизм маршрутизации останавливается сразу после нахождения первого совпадения, второй маршрут никогда не будет использоваться.

Обязательно помните о таком сценарии и продумывайте порядок, в котором регистрируются специальные маршруты.

Маршрутизация на существующие файлы

Механизм маршрутизации ASP.NET MVC отдает предпочтение физическим файлам, расположенным на сервере, перед “виртуальными” маршрутами, определенными в таблице маршрутов. Таким образом, запрос, сделанный в отношении физического файла, обойдет процесс маршрутизации, и механизм просто возвратит этот файл, а не будет пытаться анализировать URL и искать совпадающий маршрут. В некоторых случаях такое поведение полезно переопределить, заставив ASP.NET MVC предпринимать попытки маршрутизации для всех запросов. Это можно сделать, установив свойство `RouteCollection.RouteExistingFiles` в `false`.

Игнорирование маршрутов

В дополнение к определению маршрутов, которые отображаются на контроллеры и действия, ASP.NET MVC также позволяет определять маршруты с шаблонами URL, которые должны просто игнорироваться. Тот же самый объект `RoutesTable`, который открывает метод `MapRoute()`, также предлагает метод `IgnoreRoute()`, предназначенный для добавления специального маршрута, который сообщает механизму маршрутизации о необходимости игнорирования запросов к любому URL, соответствующему заданному шаблону.

Рассмотрим приведенный ниже фрагмент из стандартной логики маршрутизации, генерируемой Visual Studio. Здесь ASP.NET MVC указывается на необходимость игнорирования маршрутов, содержащих .axd – файловое расширение, которое используется для распространенных обработчиков ASP.NET, таких как `Trace.axd` и `WebResource.axd`:

```
routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
```

Благодаря такому вызову `IgnoreRoute()`, запросы к этим URL обрабатываются как нормальные запросы к ASP.NET, а не посредством механизма маршрутизации.

Метод `IgnoreRoute()` можно использовать также для игнорирования других запросов. Например, предположим, что имеется сценарий, при котором раздел веб-сайта содержит код, написанный с применением другой инфраструктуры или языка, и он не должен обрабатываться исполняющей средой ASP.NET MVC. В таком случае можно воспользоваться следующим фрагментом, чтобы сообщить ASP.NET MVC о необходимости игнорирования URL, которые начинаются с `php-app`:

```
routes.IgnoreRoute("php-app/{*pathInfo}");
```

Обратите внимание, что внутри приложения важно помещать вызовы метода `IgnoreRoute()` перед добавлением стандартных маршрутов с помощью метода `MapRoute()`.

Универсальные маршруты

Еще одной особенностью механизма разбора URL в ASP.NET MVC является возможность указания “универсального заполнителя”. Универсальные заполнители создаются за счет помещения символа звездочки (*) в начало заполнителя и могут быть включены только в качестве последнего сегмента маршрута.

В действительности вы уже видели пример универсального заполнителя в предшествующем фрагменте с `php-app`:

```
routes.IgnoreRoute("php-app/{*pathInfo}");
```

Универсальные заполнители могут также использоваться в нормальных отображенных маршрутах. Например, универсальный заполнитель можно применить в сценарии поиска для сбора низкоуровневых поисковых терминов:

```
routes.MapRoute(
    "Default",
    "{controller}/{action}/{*queryValues}",
    new { controller = "Store", action = "Search" }
);
```

В этом маршруте используются нормальные заполнители контроллера и действия, но также добавлен универсальный заполнитель для захвата всего, что следует после порций контроллера и действия в URL. Таким образом, если URL запроса выглядит как `http://demo.com/store/search/wagon/RadioFlyer`, то механизм маршрутизации разберет его, как показано в табл. 14.2.

Итак, если пользователь вводит URL вида `http://demo.com/store/search/wagon/RadioFlyer`, то механизм маршрутизации разбирает заполнители `{controller}` и `{action}`, а затем присваивает любой другой контент, находящийся после заполнителя `{action}`, единственному ключу в словаре `RouteData` по имени `queryValues`:

Таблица 14.2. Разбор универсального заполнителя

Параметр	Значение
<code>{controller}</code>	<code>store</code>
<code>{action}</code>	<code>search</code>
<code>{queryValues}</code>	<code>wagon/RadioFlyer</code>

Универсальный параметр может также применяться для того, чтобы заставить механизм маршрутизации игнорировать любые запросы, которые содержат специфическое файловое расширение. Например, если механизм маршрутизации должен игнорировать любые запросы к файлу ASPX, можно использовать универсальный маршрут, подобный показанному ниже:

```
routes.IgnoreRoute("{*allaspx}", new { allaspx=@".*\.aspx(/.*)?"});
```

В этом случае применяется перегруженная версия метода `IgnoreRoute()`, которая принимает игнорируемый URL и набор выражений, указывающий значения для параметра URL. Игнорируемый URL – это универсальный URL, который по существу

говорит о необходимости оценки каждого URL запроса, в то время как выражение, присваиваемое параметру URL, представляет собой регулярное выражение, определяющее, содержит ли запрос файл с расширением .aspx.

Ограничения маршрутов

До сих пор мы показывали, как можно создавать маршруты в приложении, и демонстрировали различные способы конструирования URL, используя заполнители, но приведенные до этого момента маршруты не ограничивали значения, которые могут вводить пользователи. Это означает, что если URL должен содержать, к примеру, только целочисленное значение (int), пользователь вполне может указать значение другого типа, приводя к отказу во время привязки модели для строго типизированных параметров действий, которые ожидают значение типа int.

К счастью, механизм маршрутизации поддерживает способ добавления проверки достоверности в отношении значений заполнителей, который называется *ограничениями маршрутов*.

Метод MapRoute() имеет переопределение, позволяющее устанавливать ограничения в заполнителях внутри маршруты. Во многом подобно установке стандартных значений заполнителей, установка ограничений сводится просто к созданию нового анонимного типа. В приведенном ниже примере с помощью простого регулярного выражения ограничиваются значения для заполнителя id:

```
routes.MapRoute(
    "Default",
    "{controller}/{action}/{id}",
    new { controller = "Home", action = "Index", id = UrlParameter.Optional },
    new { id = "(|Ford|Toyota|Honda)" }
);
```

В следующем примере значения id ограничиваются только числами:

```
routes.MapRoute(
    "Default",
    "{controller}/{action}/{id}",
    new { controller = "Home", action = "Index", id = UrlParameter.Optional },
    new { id = "\d+" }
);
```

А здесь значения id ограничиваются числами, содержащими только три цифры:

```
routes.MapRoute(
    "Default",
    "{controller}/{action}/{id}",
    new { controller = "Home", action = "Index", id = UrlParameter.Optional },
    new { id = "\d{3}" }
);
```

За счет применения регулярных выражений для определения ограничений появляются большие возможности управления значениями заполнителей.

Если ограничение не удовлетворено маршрутом, механизм маршрутизации считает, что этот маршрут не дает совпадения, и продолжает проход по таблице маршрутов в поисках соответствия. Зная это, ограничения можно использовать в сценариях, где имеются идентичные URL маршрутов, которые нужно направить на разные контроллеры или действия.

В приведенном ниже коде определены два маршрута, которые имеют идентичные URL, но разные ограничения:

```
routes.MapRoute(
    "noram",
    "{controller}/{action}/{id}",
    new { controller = "noram", action = "Index", id = UrlParameter.Optional },
    new { id = "(us|ca)" }
);

routes.MapRoute(
    "europe",
    "{controller}/{action}/{id}",
    new { controller = "europe", action = "Index", id = UrlParameter.Optional },
    new { id = "(uk|de|es|it|fr|be|nl)" }
);
```

Наконец, в то время как большинство сценариев с выполнением проверки достоверности может быть охвачено с помощью ограничения в виде регулярного выражения, бывают ситуации, когда требуются более сложные процедуры проверки. В таких случаях можно воспользоваться интерфейсом `IRouteConstraint` и создать специальное ограничение.

Как показано в следующем коде, интерфейс `IRouteConstraint` имеет единственный метод по имени `Match()`, который должен быть реализован:

```
public class CustomerConstraint : IRouteConstraint
{
    public bool Match(HttpContextBase httpContext, Route route,
                      string parameterName, RouteValueDictionary values,
                      RouteDirection routeDirection)
    {
        var cdx = new UsersDataContext();
        // Выполнить поиск в базе данных.
        var result = (from u in cdx.Users
                     where u.Username = values["user"]
                     select u).FirstOrDefault();
        return result != null;
    }
}
```

Этот код демонстрирует, каким образом можно создать более сложное ограничение маршрута, которое выполняет запрос для проверки существования в базе данных предоставленного значения. В следующем коде показано, как использовать это специальное ограничение при создании маршрута:

```
routes.MapRoute(
    "Default",
    "{controller}/{action}/{id}",
    new { controller = "Home", action = "Index" },
    new { id = new CustomerConstraint() }
);
```

Имейте в виду, что хотя это вполне допустимый сценарий использования для ограничения маршрута, который может часто встречаться, в реальных ситуациях может потребоваться принимать во внимание влияние на производительность со стороны поиска значения при запросе к приложению.

Также важно упомянуть, что существует несколько проектов с открытым кодом, которые предлагают готовые ограничения маршрутов, предоставляющие возможности, которые выходят за рамки обеспечиваемых простыми регулярными выражениями; это значит, что не придется писать собственные ограничения подобного рода. Одним из таких проектов является ASP.NET MVC Extensions (<http://mvceextensions.codeplex.com/>), включающий ряд ограничений маршрутов, таких как Range, Positive Int/Long, Guid и Enum.

Исследование маршрутов с использованием Glimpse

Поскольку маршрутизация добавляет уровень косвенности к приложению, отладка проблем с маршрутами может оказаться несколько сложной. Полезным инструментом для просмотра информации, связанной с маршрутами, во время выполнения является Glimpse.

Средство Glimpse включает вкладку Routes (Маршруты), на которой отображаются не только зарегистрированные маршруты, но также и множество другой информации, в том числе: маршрут, давший совпадение для загрузки текущей страницы; стандартные значения и ограничения для определенных маршрутов; и действительные значения для заполнителей маршрутов. Вкладка Routes в Glimpse показана на рис. 14.1.

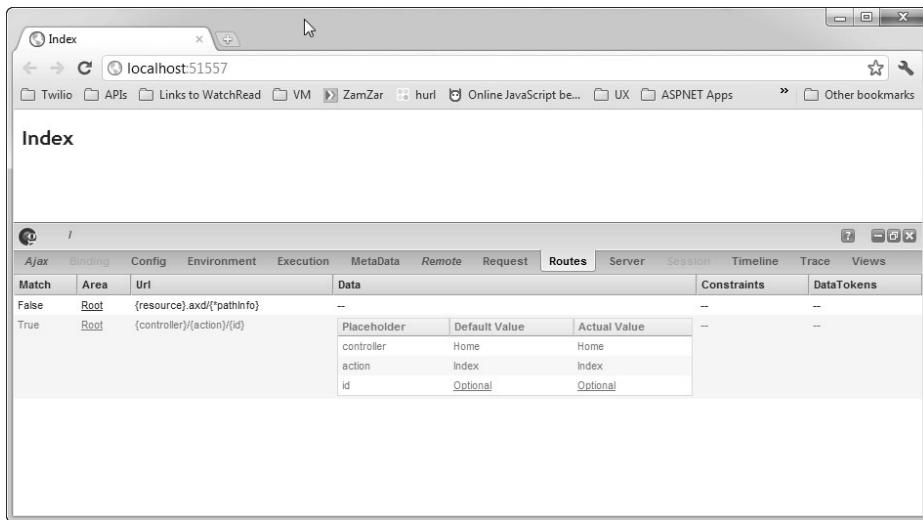


Рис. 14.1. Вкладка Routes в Glimpse

Маршрутизация на основе атрибутов

Использование метода `MapRoute()` является простым способом регистрации маршрутов для приложения, однако ему присущи некоторые недостатки. Код регистрации маршрутов изолирован от действительных контроллеров и действий, на которые маршруты, в конечном счете, отображаются, поэтому в крупных приложениях это может создать сложности при сопровождении.

Один из способов обойти указанную проблему заключается в применении приема под названием *маршрутизация на основе атрибутов*. Эта простая технология построена поверх механизма маршрутизации, комбинируя его со стандартными атрибутами

.NET, и она позволяет применять специфические маршруты прямо к действиям, снабжая их специальными атрибутами. Чтобы посмотреть, как работает данный подход, давайте создадим простой атрибут маршрута.

Нам понадобится построить две порции инфраструктуры:

- класс атрибута;
- класс, который генерирует новые маршруты из этих атрибутов.

Для начала создадим новый класс `RouteAttribute`, производный от `System.Attribute`:

```
[AttributeUsage(AttributeTargets.Method, Inherited = true, AllowMultiple = true)]
public class RouteAttribute : Attribute
{
    /// <summary>
    /// Объект JSON, содержащий ограничения части данных маршрута.
    /// </summary>
    public string Constraints { get; set; }

    /// <summary>
    /// Объект JSON, содержащий стандартные значения части данных маршрута.
    /// </summary>
    public string Defaults { get; set; }

    /// <summary>
    /// Шаблон маршрутизации URL, включающий заполнители части данных маршрута.
    /// </summary>
    public string Pattern { get; set; }

    public RouteAttribute(string pattern)
    {
        Pattern = pattern;
    }
}
```

Класс `RouteAttribute` открывает несколько простых свойств для определения URL маршрута, стандартных значений заполнителей и ограничений заполнителей. Это и все, что нужно сделать: порция, касающаяся атрибута, в примере маршрутизации на основе атрибутов готова.

Чтобы использовать созданный атрибут, необходимо просто декорировать действие в контроллере:

```
[Route("auctions/{key}-{title}/bids")]
public ActionResult Auctions(string key, string title)
{
    // Извлечь и возвратить аукционный товар.
}
```

В этом случае атрибут `Route` просто определяет шаблон маршрутизации, который должен отображаться на это действие контроллера. Можно использовать другие свойства атрибута, чтобы установить стандартные значения заполнителей или ограничения заполнителей. Можно даже применить более одного атрибута `Route`, чтобы отобразить множество маршрутов на одно и то же действие.

Далее понадобится предусмотреть для приложения способ превращения примененного атрибута `RouteAttribute` в реальные регистрацию маршрутов. Для этого мы построим новый класс по имени `RouteGenerator`.

Конструктор класса RouteGenerator будет требовать передачи некоторого количества параметров, включая экземпляр RouteCollection, текущий RequestContext и коллекцию всех действий контроллеров в приложении. В конструкторе также будет создаваться новый экземпляр объекта JavaScriptSerializer, который позволит сериализовать и десериализовать объекты JSON:

```
public RouteGenerator(
    RouteCollection routes, RequestContext requestContext,
    ControllerActions controllerActions
)
{
    _routes = routes;
    _controllerActions = controllerActions;
    _requestContext = requestContext;
    _javaScriptSerializer = new JavaScriptSerializer();
}
```

Теперь необходим метод, который генерирует новые маршруты:

```
public virtual IEnumerable<RouteBase> Generate()
{
    IEnumerable<Route> customRoutes =
        from controllerAction in _controllerActions
        from attribute in controllerAction.Attributes.OfType<RouteAttribute>()
        let defaults = GetDefaults(controllerAction, attribute)
        let constraints = GetConstraints(attribute)
        let routeUrl = ResolveRoute(attribute, defaults)
        select new Route(routeUrl, defaults, constraints, new MvcRouteHandler());

    return customRoutes;
}
```

Метод Generate() принимает список действий контроллеров и использует запрос LINQ для выбора тех из них, которые помечены атрибутом RouteAttribute, регистрируя новый маршрут для каждого действия. Для этого в запросе LINQ применяется множество вспомогательных методов, работа которых заключается в извлечении свойств, URL, стандартных значений и ограничений атрибута с последующим преобразованием их в значения, воспринимаемые новым экземпляром маршрута.

В примере 14.1 приведен полный код класса RouteGenerator.

Пример 14.1. Класс RouteGenerator

```
public class RouteGenerator
{
    private readonly RouteCollection _routes;
    private readonly RequestContext _requestContext;
    private readonly JavaScriptSerializer _javaScriptSerializer;
    private readonly ControllerActions _controllerActions;

    public RouteGenerator(
        RouteCollection routes, RequestContext requestContext,
        ControllerActions controllerActions
    )
    {
        Contract.Requires(routes != null);
        Contract.Requires(requestContext != null);
        Contract.Requires(controllerActions != null);
    }
}
```

```

        _routes = routes;
        _controllerActions = controllerActions;
        _requestContext = requestContext;
        _javaScriptSerializer = new JavaScriptSerializer();
    }

    public virtual IEnumerable<RouteBase> Generate()
    {
        IEnumerable<Route> customRoutes =
            from controllerAction in _controllerActions
            from attribute in controllerAction.Attributes.OfType<RouteAttribute>()
            let defaults = GetDefaults(controllerAction, attribute)
            let constraints = GetConstraints(attribute)
            let routeUrl = ResolveRoute(attribute, defaults)
            select new Route(routeUrl, defaults, constraints, new MvcRouteHandler());
        return customRoutes;
    }

    private RouteValueDictionary GetDefaults(
        ControllerAction controllerAction,
        RouteAttribute attribute
    )
    {
        var routeDefaults = new RouteValueDictionary(new {
            controller = controllerAction.ControllerShortName,
            action = controllerAction.Action.Name,
        });
        if (string.IsNullOrWhiteSpace(attribute.Defaults) == false)
        {
            var attributeDefaults =
                _javaScriptSerializer.Deserialize<IDictionary<string, object>>(
                    attribute.Defaults);
            foreach (var key in attributeDefaults.Keys)
            {
                routeDefaults[key] = attributeDefaults[key];
            }
        }
        return routeDefaults;
    }

    private RouteValueDictionary GetConstraints(RouteAttribute attribute)
    {
        var constraints =
            _javaScriptSerializer.Deserialize<IDictionary<string, object>>(
                attribute.Constraints ?? string.Empty);
        return new RouteValueDictionary(constraints ?? new object());
    }

    private string ResolveRoute(
        RouteAttribute attribute,
        RouteValueDictionary defaults
    )
    {
        // ЯВНЫЙ URL покрывает все.
        string routeUrl = attribute.Pattern;

```

```
// Если routeUrl не существует, попробовать его вывести.  
if (string.IsNullOrEmpty(routeUrl))  
    routeUrl = _routes.GetVirtualPath(_requestContext, defaults).VirtualPath;  
if ((routeUrl ?? string.Empty).StartsWith("/"))  
    routeUrl = routeUrl.Substring(1);  
return routeUrl;  
}  
}
```

Наконец, потребуется привязать класс `RouteGenerator`, чтобы он имел возможность запускаться и регистрировать все маршруты во время старта приложения. Для этого в методе `RegisterRoutes()` создается новый экземпляр `RouteGenerator`, которому указывается на необходимость генерации всех маршрутов, а затем осуществляется проход в цикле по маршрутам со вставкой каждого из них в `RouteTable`:

```
var routeGenerator = new RouteGenerator(routes,  
    HttpContext.Current.Request.RequestContext,  
    ControllerActions.Current);  
  
var actionroutes = routeGenerator.Generate();  
foreach (var route in actionroutes)  
{  
    RouteTable.Routes.Insert(0, route);  
}
```

Имея `RouteGenerator`, маршруты теперь можно регистрировать, просто декорируя действия посредством атрибутов, вместо того, чтобы изолировать регистрацию от действий. Очевидно, что показанный здесь класс `RouteGenerator` – лишь один из способов достичь этого. В действительности доступен ряд хороших проектов с открытым кодом, которые переносят регистрацию маршрутов на основе атрибутов на новый качественный уровень, поэтому если вы не хотите заниматься собственной реализацией, можете добавить одну из таких библиотек в свое приложение.

Расширение маршрутизации

К этому моменту вы должны хорошо понимать, каким образом создавать маршруты в приложении ASP.NET MVC, и, скорее всего, сведения, изученные до сих пор – это все что нужно. Тем не менее, временами встроенных возможностей ASP.NET MVC просто недостаточно.

К счастью, инфраструктура спроектирована так, что существует несколько точек расширения, включая и сам механизм маршрутизации. В этом разделе мы рассмотрим некоторые низкоуровневые детали конвейера маршрутизации и взглянем на предлагаемые им точки расширения.

Конвейер маршрутизации

Начнем с более подробного анализа конвейера ASP.NET MVC (рис. 14.2).

Первым делом, входящий запрос обрабатывается модулем `UrlRoutingModule`, который отвечает за соответствие запрошенного URL маршруту в приложении. Если этот модуль находит совпадение, он создает экземпляр `IRouteHandler`, ассоциируемый с маршрутом. По умолчанию в ASP.NET MVC это экземпляр класса `MvcRouteHandler`. Интерфейс `IRouteHandler` отвечает за возврат экземпляра HTTP-обработчика, который действительно обрабатывает входящий запрос.

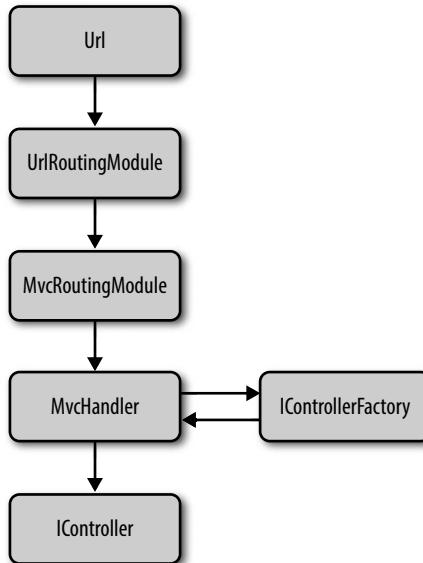


Рис. 14.2. Конвейер ASP.NET MVC

Возможно, вы уже догадались, что в ASP.NET стандартным интерфейсом `IRouteHandler` является `MvcRouteHandler`, а стандартным HTTP-обработчиком, который он создает, будет `MvcHandler`. Для создания нового экземпляра корректного класса контроллера этот HTTP-обработчик использует фабрику контроллеров. Поразительно в этом конвейере то, что вдоль всего пути предусмотрены точки расширения. Вполне возможно создать класс, производный от `UrlRoutingModule`, для добавления дополнительной функциональности к поведению отображения маршрутов либо для подключения собственной реализации интерфейсов `IRouteHandler` или `IHttpHandler`.

Давайте взглянем на простой специальный класс обработчика маршрутов:

```

public class SimpleRouteHandler : IRouteHandler
{
    public IHttpHandler GetHttpHandler(HttpContext requestContext)
    {
        return new SimpleHandler(requestContext);
    }
}

```

Как показано в этом примере, интерфейс `IRouteHandler` имеет единственный метод по имени `GetHttpHandler()`, который должен быть реализован. Внутри этого метода можно управлять тем, какой `IHttpHandler` создается и применяется для обработки маршрута. `MvcRouteHandler` обычно создает экземпляр класса `MvcHandler`, но с помощью специального `IRouteHandler` можно создавать экземпляры собственных классов обработчиков. В приведенном выше примере создается экземпляр класса `SimpleHandler` с передачей `requestContent` его конструктору, и затем этот обработчик возвращается.

В следующем коде показано, как зарегистрировать вновь созданный обработчик `SimpleRouteHandler` для приложения:

```
routes.Add(new Route("{controller}/{action}/{id}", new SimpleRouteHandler()));
```

Как видите, один из конструкторов класса `Route` принимает в качестве параметра обработчик маршрутов. Если этот маршрут выбран модулем `UrlRoutingModule`, он будет знать о необходимости создания экземпляра `SimpleRouteHandler`, а не стандартного `MvcRouteHandler`.

А теперь давайте рассмотрим реальный пример, когда построение собственного обработчика маршрутов может быть полезным. Если вы работаете в организации, существующей на протяжении более пяти лет, то можете иметь дело с унаследованными объектами COM (Component Object Model – модель компонентных объектов), предоставляющими функциональность, от которой зависит сайт. Хотя .NET Framework облегчает обращение к компонентам COM, этот вид взаимодействия не так прост, как работа с собственными объектами .NET Framework.

Например, в исполняющей среде ASP.NET каждый запрос обрабатывается в собственном потоке, и когда приходится иметь дело с COM, каждый из этих потоков является собственным “многопоточным апартаментом” (multithreaded apartment – MTA).

Однако COM-компоненты VB6 не совместимы с многопоточными апартаментами COM в ASP.NET; таким образом, каждый раз, когда запрос ASP.NET взаимодействует с такими COM-объектами, среда COM должна выполнять эти запросы посредством “однопоточного апартамента” (single-threaded apartment – STA). Данная ситуация может привести к возникновению серьезного узкого места, т.к. каждое приложение имеет только один STA-апартамент, поэтому все запросы к COM-объектам вынуждены будут ожидать его освобождения.

Выяснить состояние апартамента для запроса очень легко с применением метода `GetApartmentState()`, как показано в следующем коде:

```
public string ThreadState()
{
    var thread = System.Threading.Thread.CurrentThread;
    ApartmentState state = thread.GetApartmentState();
    return state.ToString();
}
```

Если запустить этот метод в нормальном приложении ASP.NET, он выдаст в браузер информацию о том, что апартаментом является MTA.

Среда COM спроектирована так, что она не создает множество потоков STA на запрос. Когда поступает первый запрос, COM создает поток STA для его обработки. Затем для каждого последующего запроса вместо создания нового потока среда COM помещает вызовы в очередь того же самого потока STA.

Инфраструктура ASP.NET Web Forms предлагает простое решение: страничную директиву `AspCompat`. Если установить директиву `AspCompat` в `true` на странице Web Forms, то наряду с созданием и использованием пула потоков STA среда COM получает доступ к объектам запроса и ответа страницы. Это также позволяет COM-объектам, которые зарегистрированы с `ThreadingModel="Apartment"`, быть созданными в апартаментах их создателей, при условии, что эти создатели сами функционируют в STA-апартаментах. Преимущество такого подхода в том, что поскольку COM-объект разделяет апартамент с создателем, множество запросов теперь могут выполняться параллельно без всяких узких мест.

К сожалению, ASP.NET MVC не предлагает ничего такого, что работало бы подобно `AspCompat`, поэтому вы должны реализовать собственное решение для имитации поведения директивы `AspCompat`. Это великолепная возможность создать собственный обработчик маршрутов, который позволил бы потоку запроса выполнятся в апартаменте STA, а не MTA.

Давайте посмотрим, как можно создать некоторую специальную логику, используя точки расширения ASP.NET MVC, для имитации директивы AspCompat в приложении ASP.NET MVC. Мы начнем с создания нового обработчика маршрутов, который перенаправляет входящие URL запросов на специальный класс HTTP-обработчика:

```
public class AspCompatHandler : IRouteHandler
{
    protected IHttpHandler GetHttpHandler(RequestContext requestContext)
    {
        return new AspCompatHandler(requestContext);
    }
}
```

Как было показано ранее, в классе AspCompatHandler мы используем метод GetHttpHandler(), чтобы предоставить ASP.NET MVC экземпляр AspCompatHandler, который должен применяться для маршрутизации входящих запросов.

Далее можно создать AspCompatHandler. Для этого мы начинаем с создания нового класса, производного от System.Web.UI.Page. За счет наследования от стандартного класса Page из Web Forms мы можем получить доступ к директиве AspCompat, определенной ASP.NET:

```
public class AspCompatHandler : System.WebForms.UI.Page
{
    public AspCompatHandler(RequestContext requestContext)
    {
        this.RequestContext = requestContext;
    }

    public RequestContext RequestContext { get; set; }

    protected override void OnInit(EventArgs e)
    {
        string requiredString =
            this.RequestContext.RouteData.GetRequiredString("controller");
        var controllerFactory = ControllerBuilder.Current.GetControllerFactory();
        var controller =
            controllerFactory.CreateController(this.RequestContext, requiredString);
        if (controller == null)
            throw new InvalidOperationException("Could not find controller: " +
                requiredString);

        try
        {
            controller.Execute(this.RequestContext);
        }
        finally
        {
            controllerFactory.ReleaseController(controller);
        }
        this.Context.ApplicationInstance.CompleteRequest();
    }
}
```

В этом классе мы переопределяем метод OnInit(), чтобы добавить код, который ищет и выполняет контроллер, предназначенный для обработки запроса. Это имитирует базовое поведение стандартного класса MvcHandler, который обычно используется для маршрутизации запросов. Мы также собираемся переопределить метод ProcessRequest() класса Page и гарантировать, что он не будет случайно вызван.

После того, как базовый класс маршрута создан, понадобится добавить к нему реализацию интерфейса `IHttpAsyncHandler`. Реализация этого интерфейса – это то, что позволяет ASP.NET MVC применять класс `AspCompatHandler` в качестве обработчика маршрутов. Интерфейс `IHttpAsyncHandler` имеет два метода, которые должны быть реализованы – `BeginProcessRequest()` и `EndProcessRequest()`; мы будем использовать их для сообщения ASP.NET о необходимости обработки запроса, используя директиву `AspCompat`:

```
public IAsyncResult BeginProcessRequest(HttpContext context, AsyncCallback cb,
                                         object extraData)
{
    return this.AspCompatBeginProcessRequest(context, cb, extraData);
}

public void EndProcessRequest(IAsyncResult result)
{
    this.AspCompatEndProcessRequest(result);
}
```

Легко заметить, что с помощью методов `IHttpAsyncHandler` мы просто передаем запрос методу `AspCompatBeginProcessRequest()` и затем получаем результат из метода `AspCompatEndProcessRequest()` (оба они представлены классом `Page`).

Теперь, когда написан обработчик маршрутов, осталось только присоединить его к определению маршрута:

```
context.MapRoute("AspCompatRoute", "{controller}/{action}",
                  new { controller = "Home", action = "Index" }
                  ).RouteHandler = new AspCompatHandler();
```

Если вы запустите приведенный ранее код для извлечения текущего состояния `ApartmentState`, то увидите, что контроллер теперь выполняется в режиме STA.

И поскольку запросы функционируют в потоках STA, среда COM не должна создавать собственные потоки STA, чтобы выполнять обращения к компонентам COM. Более того, учитывая, что компоненты COM находятся в STA-апартаментах их создателей, все они могут выполняться независимо друг от друга, делая возможным настоящий параллелизм.

Резюме

В этой главе вы ознакомились с главными концепциями маршрутов и маршрутизации в ASP.NET MVC Framework. Глава начиналась с объяснения причин, по которым URL приложения настолько важны, как с точки зрения удобства использования, так и с точки зрения поисковой оптимизации.

Затем было показано, как создавать новые маршруты с применением метода `RouteTable.MapRoute()`, который добавляет маршруты в статический словарь `RouteTable`, а также как сформировать различные структуры URL маршрута.

Далее вы узнали, каким образом создавать и использовать ограничения маршрутов для управления значениями, которые пользователи могут отправлять приложению через маршруты, а также как строить специальные ограничения для приложения.

Наконец, был продемонстрирован альтернативный путь создания маршрутов с применением подхода на основе атрибутов и вдобавок некоторые точки расширения в конвейере маршрутизации ASP.NET MVC. Теперь вы должны хорошо понимать работу маршрутизации в ASP.NET MVC Framework.

Многократно используемые компоненты пользовательского интерфейса

До сих пор вы видели множество вариантов построения многократно используемых компонентов для приложения ASP.NET MVC. Однако эти варианты позволяют создавать представления или действия, которые могут повторно использоваться только внутри одного проекта. Другими словами, они являются больше “разделяемыми” компонентами, чем действительно “многократно используемыми”, поскольку их невозможно применять за пределами проекта, не прибегнув к приему “повторного использования кода” (который известен также как “копирование/вставка”).

В этой главе вы узнаете, как создавать по-настоящему многократно используемые компоненты, которые могут служить в качестве библиотеки для разных проектов.

Что ASP.NET MVC предлагает в готовом виде

Перед тем как переходить к созданию межпроектных многократно используемых компонентов, давайте кратко рассмотрим, что инфраструктура ASP.NET MVC предлагает в готовом виде.

Частичные представления

Частичные представления позволяют создавать многократно используемый контент. Чтобы оставаться по-настоящему многократно используемыми, частичные представления должны содержать мало функциональной логики или вообще не содержать ее, т.к. они представляют собой модульные единицы компоновки в более крупных представлениях. Частичные представления – это файлы, которые имеют то же самое расширение .cshtml или .vbhtml, но находятся в папке /Views/Shared/. Для их визуализации применяется синтаксис @Html.Partial("_имяЧастичногоПредставления"), как показано ниже:

```
@Html.Partial("_Auction")
```

Расширения класса HtmlHelper или специальные вспомогательные методы HTML

Специальные вспомогательные методы HTML – это расширяющие методы, применяемые к классу HtmlHelper, который может использоваться в представлениях для вывода чистой HTML-разметки. Они следуют тем же самым правилам, что и частичные представления – т.е. не имеют функциональной логики и представляют небольшую единицу компоновки, – но являются в большей степени специализированными. Распространенным примером может служить расширение HTMLHelper, визуализирующее текстовое поле вместе с соответствующей меткой (такая комбинация часто применяется для удобства ввода):

```
@Html.TextBoxAccessible("FirstName", @Model.FirstName)
```

Ниже приведен код этого расширения:

```
public static class HtmlHelperExtensions
{
    public static HtmlString TextBoxAccessible(this HtmlHelper html,
                                              string id, string text)
    {
        return new HtmlString(html.Label(id)
                               + html.TextBox(id, text).ToString());
    }
}
```

Шаблоны отображения и редактирования

Шаблоны отображения и редактирования были введены в ASP.NET MVC 2 и позволяют создавать строго типизированные представления наподобие следующего:

```
@Html.DisplayFor(model => model.Product)
```

Шаблоны отображения и редактирования – это частичные представления, расположенные в подпапках DisplayTemplates или EditorTemplates папки контроллера (или папки Views\Shared). Например, если создать частичное представление по имени Product.cshtml в Views\Shared\DisplayTemplates или Views\Product\DisplayTemplates с разметкой для отображения информации о товаре (Product) каким-либо образом, то @Html.DisplayFor(model => model.Product) будет использовать этот шаблон DisplayTemplate для визуализации Product:

```
@model Product
@if (Model != null) {
    <!-- Разметка для визуализации сведений о товаре --&gt;
}</pre>
```

В случае несоответствия типа шаблону будет применяться представление .ToString() объекта.

Поскольку шаблоны часто привязаны к конкретной модели, они могут содержать определенное количество бизнес-логики. Будучи строго типизированными, они также помогают перехватывать ошибки на этапе компиляции, а не во время выполнения.

Html.RenderAction()

Вспомогательный метод RenderAction() выполняет действие контроллера и затем вставляет HTML-вывод в родительское представление. По этой причине

`RenderAction()` позволяет многократно использовать функциональную логику, а также компоновку. Этот вспомогательный метод HTML применяется, когда компоновка является сложной, и часто в случае, если необходимо обеспечить повторное использование бизнес-логики.

Продвижение на шаг вперед

Варианты, которые обсуждались выше, хорошо работают, когда требуется многократное использование компонента в рамках одного приложения. Множество действий контроллера могут ссылаться на одно и то же представление, когда это разделяемое представление находится в той же папке, что и другие представления для данного контроллера, или хранится в папке `Shared`, а разные представления могут внутри себя вызывать специальные вспомогательные методы для многократного использования логики презентации. Но как разделять представления/компоненты между проектами?

В мире ASP.NET Web Forms этого можно достичь, создав пользовательские элементы управления или специальные элементы управления, которые могут быть скомпилированы в автономные сборки. Такие сборки могут распределяться между проектами, тем самым допуская их многократное использование в этих проектах.

Механизм представлений Web Forms предлагает класс `ViewUserControl`, который может применяться для создания подобных компонентов, предназначенных для инфраструктуры MVC. Тем не менее, механизм представлений Razor не предоставляет такого метода. В этом разделе мы покажем, как достигнуть чего-то аналогичного, используя API-интерфейс Razor.

Генератор одиночных файлов Razor

Представления Razor – это в действительности причудливые визуальные конструкторы, которые, в конечном счете, генерируют код .NET. Генерируемый код может быть скомпилирован в сборки, а скомпилированные сборки, конечно же, допускают многократное использование между проектами. Таким образом, вам необходим инструмент, который может взять представления Razor, созданные в отдельном проекте, и запустить API-интерфейс Razor прямо в отношении них для генерации кода .NET. Хотя настоящая глава дает всю информацию, нужную для построения такого инструмента, важно отметить, что делать это не понадобится. Подходящий инструмент уже имеется в сообществе открытого кода.

Установка генератора одиночных файлов Razor

Несмотря на то что исходный код находится на сайте CodePlex, установщик генератора одиночных файлов Razor (*Razor Single File Generator*) доступен в галерее расширений Visual Studio, поэтому проще всего установить его оттуда. Откройте диспетчер расширений Visual Studio (через пункт меню `Tools`⇒`Extension Manager...` (`Сервис`⇒`Диспетчер расширений...`)) и найдите в онлайновой галерее (Online Gallery) элемент `Razor Generator` (Генератор Razor), как показано на рис. 15.1.

После того, как генератор Razor установлен (не забудьте перезапустить Visual Studio!), создайте новый проект для размещения разделяемых представлений. Помимо того факта, что вы будете применять специальный инструмент к файлам представлений, ничего специфического с этим проектом больше не связано. Просто создайте новый проект `Class Library` (Библиотека классов), как показано на рис. 15.2, в рабочем решении и назовите его `ReusableComponents`.

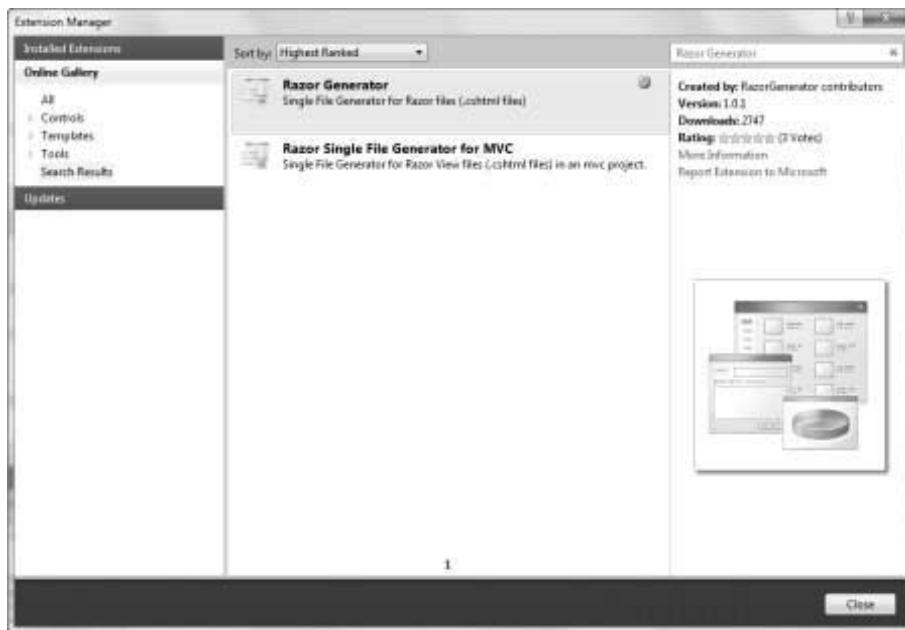


Рис. 15.1. Установка генератора Razor в диспетчере расширений

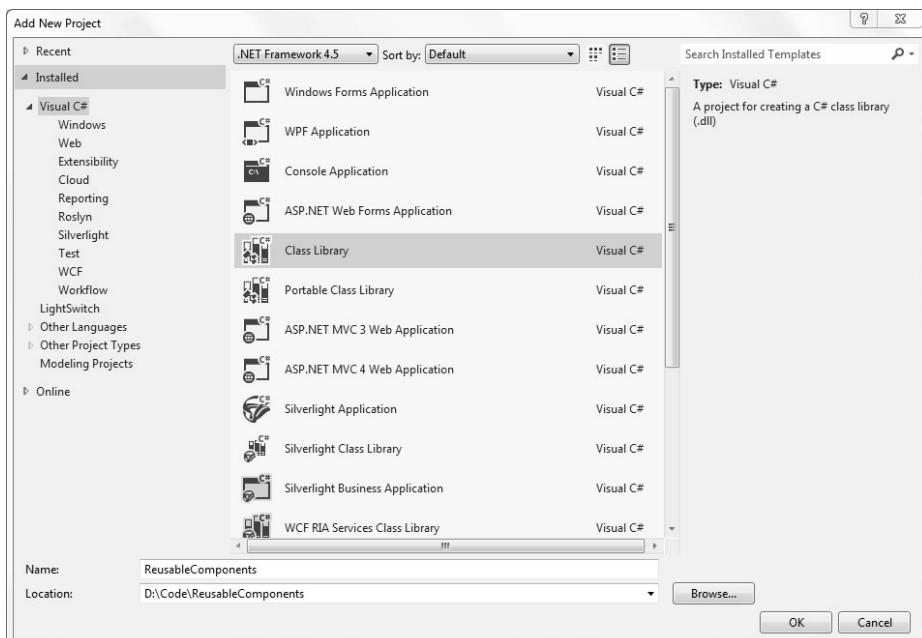


Рис. 15.2. Создание нового проекта Class Library для многократно используемых представлений

Создание многократно используемых представлений ASP.NET MVC

Одним из наиболее широко распространенных сценариев для представления, разделяемого между проектами, является страница обобщенного сообщения об ошибке. Итак, давайте создадим ее и посмотрим, как генератор одиночных файлов Razor обрабатывает представления ASP.NET MVC.

Создание многократно используемых представлений ASP.NET MVC с помощью генератора одиночных файлов Razor в основном похоже на создание представлений внутри самого проекта ASP.NET MVC. При создании структуры папок `~/Views`, которая соответствует соглашению, принятому в ASP.NET MVC, единственное, что придется сделать – это ассоциировать представления с генератором одиночных файлов Razor, установив свойство *Custom Tool* (Специальный инструмент) каждого представления в `RazorGenerator`.

Поскольку новая библиотека классов `ReusableComponents` не является проектом ASP.NET MVC, папка `~/Views` отсутствует, поэтому создайте ее. Новое представление, которые вы собираетесь добавить, будет использоваться множеством контроллеров, так что структура папок библиотеки классов должна отражать это: создайте непосредственно под `~/Views` еще одну папку и назовите ее `Shared`, следуя соглашению о структуре папок приложения ASP.NET MVC. По завершении библиотека классов `ReusableComponents` должна выглядеть примерно так, как показано на рис. 15.3.

Теперь, имея структуру папок, добавьте в папку `Shared` новый файл по имени `GenericError.cshtml`, щелкнув на этой папке правой кнопкой мыши и выбрав в контекстном меню пункт `Add⇒New Item...` (Добавить⇒Новый элемент...). Так как это проект `Class Library`, а не `ASP.NET MVC`, среда Visual Studio откажется показывать элемент типа `MVC 4 View Page (Razor)` (Страница представления MVC 4 (Razor)). Это нормально; просто выберите другой тип элементов простого контента, такой как `Text File` (Текстовый файл) или `HTML Page` (HTML-страница). Поскольку новый элемент (`GenericError.cshtml`) имеет файловое расширение `.cshtml`, среда Visual Studio будет знать, что это шаблон Razor.

Хотя Visual Studio распознает новый файл как шаблон Razor, генератору одиночных файлов Razor необходимо сообщить о необходимости начать генерацию кода именно с этого шаблона. Чтобы привязать генератор, откройте окно свойств файла `GenericError.cshtml` и установите свойство *Custom Tool* в `RazorGenerator`. Корректно сконфигурированный генератор Razor показан на рис. 15.4.

Замените содержимое нового файла `GenericError.cshtml` следующей разметкой Razor:

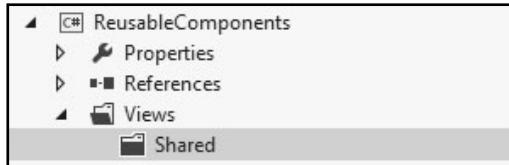


Рис. 15.3. Проект `ReusableComponents` со структурой папок `~/Views`

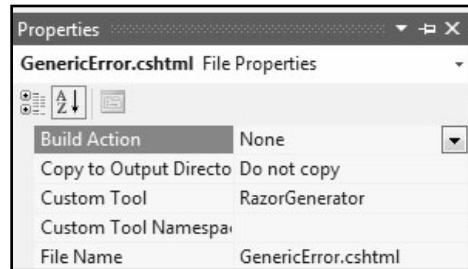


Рис. 15.4. Установка свойства *Custom Tool* в `RazorGenerator`

```

@{ Layout = null; }
<html>
<head>
    <title>Website Error!</title>
    <style>
        body { text-align: center; background-color: #6CC5C3; }
        .error-details .stack-trace { display: none; }
        .error-details:hover .stack-trace { display: block; }
    </style>
</head>
<body>
    <h2>We're sorry, but our site has encountered an error!</h2>
    
    @if (ViewData["ErrorMessage"] != null) {
        <div class="error-details">
            <h2>@ViewData["ErrorMessage"]</h2>
            <div class="stack-trace">@ViewData["StackTrace"]</div>
        </div>
    }
</body>
</html>

```

Немедленно после установки свойства Custom Tool вы должны увидеть, что генератор одиночных файлов Razor построил класс `GenericError.cs`, располагающийся под `GenericError.cshtml` (рис. 15.5).

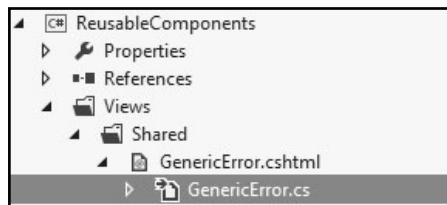


Рис. 15.5. Новый файл, созданный генератором Razor



Если вы не видите сгенерированного файла, значит, что-то пошло не так. Удостоверьтесь, что правильно указали имя специального инструмента (`RazorGenerator`, без пробелов). Если все по-прежнему не работает, повторите шаги, приведенные в начале раздела. Не забудьте, что после установки генератора Razor среда Visual Studio должна быть перезапущена, и проверьте все журналы установки на предмет отсутствия ошибок в процессе установки.

Откройте этот новый файл и просмотрите сгенерированное содержимое. Сгенерированный код действует подобно любому другому коду, компилируясь в сборку, которую можно разделять между любым количеством веб-сайтов.

Включение предварительно скомпилированных представлений в веб-приложение ASP.NET MVC

После выполнения шагов, приведенных в этом разделе, у вас остается проект библиотеки классов, заполненный предварительно скомпилированными представлениями ASP.NET MVC Razor. И что теперь? Из-за стандартных соглашений, принятых в механизме представлений ASP.NET MVC Razor, этот механизм не имеет возможности

находить представления за рамками стандартных путей поиска (папки Views в веб-приложении ASP.NET MVC), поэтому ему ничего не известно о существовании предварительно скомпилированных представлений, не говоря уже о том, как их выполнять.

Решение этой проблемы заключается в использовании PrecompiledMvcEngine – специального механизма представлений, построенного разработчиками генератора одиночных файлов Razor, который расширяет основной механизм представлений Razor возможностью поиска предварительно скомпилированных представлений. Простейший способ приступить к работе с PrecompiledMvcEngine – применить диспетчер пакетов NuGet для установки пакета PrecompiledMvcEngine в проект библиотеки классов, который содержит предварительно скомпилированные представления. Пакет PrecompiledMvcEngine добавляет к проекту ряд артефактов.

Несколько файлов web.config

API-интерфейс Razor и средство IntelliSense для Razor в Visual Studio предполагают, что представления Razor расположены внутри проекта веб-приложения, и читают их конфигурационную информацию из файлов web.config проекта. Даже если ваш проект является проектом библиотеки классов, файлы web.config, добавленные пакетом PrecompiledMvcEngine, предоставляют Visual Studio достаточно информации для средства Razor IntelliSense, даже относительно представлений, которые используют генератор одиночных файлов Razor.

Пример представления Razor

Пакет PrecompiledMvcEngine добавляет в папку ~/Views/Home проекта пример представления Razor по имени Test.cshtml, чтобы показать, как должны быть сконфигурированы предварительно скомпилированные представления. Если все работает правильно, вы должны увидеть, что это представление немедленно генерирует файл отдельного кода (Test.cs). Представление Test.cshtml – это просто ссылка, поэтому его можно как угодно модифицировать, переименовывать или даже полностью удалить.

~/App_Start/PrecompiledMvcViewEngineStart.cs

Хотя имя не особенно важно, файл PrecompiledMvcViewEngineStart.cs содержит логику (показанную ниже), которая сообщает приложению ASP.NET MVC о необходимости использования PrecompiledMvcEngine для всех предварительно скомпилированных представлений Razor в этом проекте библиотеки классов.

Файл PrecompiledMvcViewEngineStart.cs также включает атрибут WebActivator.PreApplicationStartMethod, который указывает библиотеке WebActivator выполнить метод PrecompiledMvcViewEngineStart.Start(), когда веб-приложение стартует, регистрируя PrecompiledMvcEngine в коллекции ViewEngines веб-приложения. Содержимое файла PrecompiledMvcViewEngineStart.cs выглядит следующим образом:

```
[assembly: WebActivator.PreApplicationStartMethod(
    typeof(ReusableComponents.App_Start.PrecompiledMvcViewEngineStart),
    "Start"
)]
public static class PrecompiledMvcViewEngineStart {
    public static void Start() {
        var currentAssembly = typeof(PrecompiledMvcViewEngineStart).Assembly;
        var engine = new PrecompiledMvcEngine(currentAssembly);
        ViewEngines.Engines.Insert(0, engine);
        VirtualPathFactoryManager.RegisterVirtualPathFactory(engine);
    }
}
```

После того как NuGet-пакет PrecompiledMvcViewEngine установлен, а файл ~/Views/Home/Index.cshtml перемещен из примера сайта-блога в проект библиотеки классов ReusableComponents, вы должны иметь возможность запустить веб-сайт и увидеть, что все работает, как это делалось ранее. Среда ASP.NET MVC теперь выполняет предварительно скомпилированный файл Index.cshtml из библиотеки классов, не заботясь о том, что этот файл не существует в локальной папке ~/Views. Но откуда PrecompiledMvcViewEngine знает, какое представление визуализировать?

Мы уже видели, что PrecompiledMvcViewEngine известно, как визуализировать предварительно скомпилированные представления Razor в приложении ASP.NET MVC, и PrecompiledMvcViewEngineStart заботится о регистрации PrecompiledMvcViewEngine для веб-приложения, так что остался только один недостающий элемент головоломки: обнаружение предварительно скомпилированного представления. Может показаться удивительным, но PrecompiledMvcViewEngine по-прежнему полагается на соглашение о папке Views из ASP.NET MVC, используя относительные пути к файлам для нахождения представлений. Однако это немного сбивает с толку. Механизм PrecompiledMvcViewEngine не обращает внимания на физические файлы – он ищет атрибут System.Web.WebPages.PageVirtualPathAttribute, который генератор одиночных файлов Razor добавляет к каждому созданному им представлению и который включает относительный путь к файлу представления.

Ниже приведено несколько первых строк из примера представления Test.cshtml, которые содержат PageVirtualPathAttribute:

```
[System.Web.WebPages.PageVirtualPathAttribute("~/Views/Home/Test.cshtml")]
public class Test : System.Web.Mvc.WebViewPage<dynamic>
```

Поскольку имя виртуального пути является относительным, независимо от того, где находится представление ~/Views/Home/Test.cshtml – в приложении ASP.NET MVC или в проекте библиотеки классов – его виртуальный путь будет одним и тем же. Таким образом, когда приложение ASP.NET MVC запрашивает представление Test в контроллере HomeController, механизм PrecompiledMvcViewEngine знает о том, что нужно применять предварительно скомпилированное представление Test.cshtml, зарегистрированное с виртуальным путем ~/Views/Home/Test.cshtml.



Удостоверьтесь, что добавляете пакет PrecompiledMvcEngine в проект библиотеки классов, который содержит предварительно скомпилированные представления, а *не* в проект веб-приложения ASP.NET MVC. Веб-приложение будет нуждаться в сборке PrecompiledMvcEngine во время выполнения, но артефакты, которые NuGet-пакет устанавливает в ваш пакет, предназначены только для проектов библиотек классов, содержащих предварительно скомпилированные представления Razor.

Создание многократно используемых вспомогательных методов ASP.NET MVC

Генератор одиночных файлов Razor можно также применить к шаблонам Razor, которые включают вспомогательные методы Razor, для получения аналогичного результата, как если бы шаблоны находились в папке App_Code приложения ASP.NET MVC.

Генератор одиночных файлов Razor ожидает, что шаблоны вспомогательных методов Razor расположены в папке ~/Views/Helpers, поэтому перед тем, как можно будет создавать любые вспомогательные методы, понадобится создать эту папку. После создания папки Helpers выполняйте те же самые шаги, которые осуществлялись

ранее для добавления файла шаблона Razor в новую папку `Helpers`. Назовите файл `TwitterHelpers.cshtml`. Затем установите свойство `Custom Tool` в `RazorGenerator`, как это делалось для шаблона представления ASP.NET MVC.

Немедленно после установки этого свойства вы должны увидеть автоматически сгенерированный файл `Twitter-Helpers.cs`. Откройте этот файл и обратите внимание, что генератор Razor успешно проанализировал пустой шаблон Razor и сгенерировал класс C#, готовый для наполнения вспомогательными методами.

Пустой класс не особенно полезен, поэтому давайте создадим вспомогательную функцию, используя стандартный синтаксис Razor:

```
@helper TweetButton(string url, string text) {
    <script src="http://platform.twitter.com/widgets.js" type="text/
javascript">
    </script>
    <div>
        <a href="http://twitter.com/share" class="twitter-share-button"
           data-url="@url" data-text="@text">Tweet</a>
    </div>
}
```

Сохранение файла и переключение обратно на сгенерированный файл `TwitterHelpers.cs` показывает, что он был обновлен снова в реальном времени. На этот раз статический вспомогательный класс содержит код для специальной вспомогательной функции `TweetButton`. В примере 15.1 приведен полный автоматически сгенерированный код (для лучшей читабельности комментарии и некоторые пробельные символы были удалены).

Пример 15.1. Автоматически сгенерированный код вспомогательного класса

```
namespace ReusableComponents.Views.Helpers
{
    using System;
    using System.Collections.Generic;
    using System.IO;
    using System.Linq;
    using System.Net;
    using System.Text;
    using System.Web;
    using System.Web.Helpers;
    using System.Web.Mvc;
    using System.Web.Mvc.Ajax;
    using System.Web.Mvc.Html;
    using System.Web.Routing;
    using System.Web.Security;
    using System.Web.UI;
    using System.Web.WebPages;

    [System.CodeDom.Compiler.GeneratedCodeAttribute("RazorGenerator", "1.1.0.0")]
    public static class TwitterHelpers
    {
        public static System.Web.WebPages.HelperResult
            TweetButton(string url, string text)
        {
            return new System.Web.WebPages.HelperResult(_razor_helper_writer => {
                WebViewPage.WriteLineTo(@_razor_helper_writer,
                    "<script src=\"http://platform.twitter.com/widgets.js\" "+
                    "type=\"text/javascript\">" + "</script>\r\n");
            });
        }
    }
}
```

```
        WebViewPage.WriteLineTo(@__razor_helper_writer,
            "<div>\r\n" +
            "<a href=\"http://twitter.com/share\" " +
            "class=\"twitter-share-button data-url=\""
        );
        WebViewPage.WriteTo(@__razor_helper_writer, url);
        WebViewPage.WriteLineTo(@__razor_helper_writer, "\" data-text=\"\"");
        WebViewPage.WriteTo(@__razor_helper_writer, text);
        WebViewPage.WriteLineTo(@__razor_helper_writer,
            "\">\r\n    >Tweet</a>\r\n" +
            "</div>\r\n"
        );
    });
}
```

Имея такой автоматически сгенерированный класс, веб-сайты ASP.NET MVC, ссылающиеся на сборку ReusableComponents, будут способны пользоваться вспомогательной функцией TweetButton в точности как любым другим вспомогательным методом, определенным в папке App Code веб-сайта.

Например:

```
@using ReusableComponents.Views.Helpers
<div>
    @TwitterHelpers.TweetButton(url, message)
</div>
```

Модульное тестирование представлений Razor

Многие рекомендуемые приемы выступают в пользу сохранения логики внутри представлений как можно более ограниченной и простой. Тем не менее, возможность выполнения модульного тестирования представлений MVC на основе Razor по-прежнему обеспечивает преимущества в некоторых сценариях.

Взглядите на фрагмент кода с примером представления ASP.NET MVC Razor:

```
<p>
    Order ID:
    <span id='order-id'>@Model.OrderID</span>
</p>
<p>
    Customer:
    @Html.ActionLink(
        @Model.CustomerName,
        "Details", "Customer",
        new { id = @Model.CustomerID },
        null)
</p>
```

Стандартный класс представления ASP.NET MVC Razor открывает свойства наподобие Model, Html и т.д., от которых зависит это представление. Таким образом, чтобы скомпилировать и выполнить представление за пределами исполняющей среды ASP.NET MVC, потребуется создать специальный базовый класс шаблона,

который также реализует эти свойства. Ниже приведен фрагмент кода из класса OrderInfoTemplateBase, модифицированный для включения свойств Model и Html, поэтому они могут использоваться для компиляции предыдущего представления:

```
public abstract class OrderInfoTemplateBase
{
    public CustomerOrder Model { get; set; }
    public HtmlHelper Html { get; set; }
}
```

Класс OrderInfoTemplateBase теперь удовлетворяет зависимостям шаблона от базовых классов ASP.NET MVC, действуя в качестве замены этих базовых классов ASP.NET MVC. Введение специальных базовых классов, таких как OrderInfoTemplateBase, обеспечивает полный контроль над свойствами и функциональностью, предоставляемыми шаблону. Специальные базовые классы также снижают необходимость в выполнении представлений ASP.NET MVC в рамках исполняющей среды ASP.NET MVC.

В примере 15.2 демонстрируется мощь замены производственных компонентов пробными объектами.

Пример 15.2. Модульный тест выполняет экземпляр шаблона Razor, используя пробные объекты

```
public void ShouldRenderLinkToCustomerDetails()
{
    var mockHtmlHelper = new Mock<HtmlHelper>();
    var order = new CustomerOrder()
    {
        OrderID = 1234,
        CustomerName = "Homer Simpson",
    };
    // Создать экземпляр и установить свойства.
    var template = (OrderInfoTemplateBase)Activator.CreateInstance(/*...*/);
    template.Html = mockHtmlHelper.Object;
    template.Model = customerOrder;
    template.Execute();
    // Проверить, что ссылка была сгенерирована.
    mockHtmlHelper.Verify(htmlHelper =>
        htmlHelper.ActionLink(
            order.CustomerName,
            "Details", "Customer",
            It.IsAny<object>()
        );
}
```

За счет замены производственного класса HtmlHelper пробной реализацией, модульный тест может легко делать утверждения в коде и проверять их удовлетворение, не полагаясь на исполняющую среду ASP.NET MVC.



Если вы применяете генератор одиночных файлов Razor для создания многократно используемых представлений, подходы на основе рефлексии, такие как Activator.CreateInstance(), не понадобятся.

Поскольку генератор одиночных файлов Razor строит действительные классы, вам нужно будет только создать экземпляр такого класса (например, var template = new CustomerOrderTemplate();) и запустить тесты на этом новом экземпляре.

Возможность внедрения пробных объектов и объектов-заглушек вместо производственных типов является весьма благоприятной для модульного тестирования. Без такой возможности большинству сайтов пришлось бы прибегать к запуску всех тестов пользовательского интерфейса посредством медленного и ненадежного тестирования на основе браузера. В противоположность этому, внедрение пробных объектов и объектов-заглушек позволяет разработчикам создавать модульные тесты, которые выполняются в считанные миллисекунды.

Резюме

Инфраструктура ASP.NET MVC предлагает множество способов создания многократно используемых компонентов. Частичные представления, шаблоны отображения и редактирования, вспомогательные классы/функции HTML и метод RenderAction() обеспечивают простые приемы повторного использования компонентов внутри одного проекта. С помощью API-интерфейса Razor также возможно создавать многократно используемые компоненты, которые могут разделяться между проектами. В этой главе было показано, как установить и работать с генератором одиночных файлов Razor для построения повторно используемых представлений. Кроме того, рассматривались вопросы модульного тестирования представлений с применением пробных объектов.

Контроль качества

В этой части...

Глава 16. Регистрация в журнале

Глава 17. Автоматизированное тестирование

Глава 18. Автоматизация построения

Регистрация в журнале

Ошибки в программном обеспечении являются жизненным фактом, независимо от того, сколько времени было потрачено на обдумывание архитектуры приложения или насколько хорошо написан код — рано или поздно, но вы столкнетесь с такими проблемами.

С целью минимизации влияния этих ошибок на сайт к обработке ошибок и регистрации в журнале необходимо относится как к любому другому важному средству: планировать их реализацию в проекте приложения на как можно более раннем этапе.

В этой главе мы рассмотрим инструменты для обработки ошибок, регистрации в журнале и мониторинга, которые можно использовать для улучшения производительности приложения и обеспечения возможности отслеживать проблемы, когда они возникают.

Обработка ошибок в ASP.NET MVC

Пока приложение занято обработкой HTTP-запроса, есть немало вещей, которые могут пойти не так, как планировалось. К счастью, ASP.NET MVC делает обработку всех таких ситуаций относительно простой.

Поскольку приложения ASP.NET MVC запускаются поверх ядра ASP.NET Framework, они имеют доступ к тем же основным возможностям инфраструктуры, что и приложения Web Forms, включая установку специальной страницы ошибок для обработки специфичных кодов состояния, когда они возникают.

Давайте посмотрим, что делать с ошибками в веб-приложении ASP.NET MVC, принудительно введя исключение в образцовое приложение Ebuy. Для этого откройте контроллер HomeController и добавьте в действие About контроллера код генерации нового исключения:

```
public ActionResult About()
{
    ViewBag.Message = "Your quintessential app description page.";
    throw new Exception("Something went wrong!");
}
```

Чтобы инициировать это исключение, просто запустите сайт и перейдите на URL вида /home/about, в результате чего должна отобразиться стандартная страница ошибок ASP.NET, показанная на рис. 16.1.

Теперь, когда сайт генерирует исключение, давайте добавим подходящую обработку ошибок, чтобы иметь с ним дело.

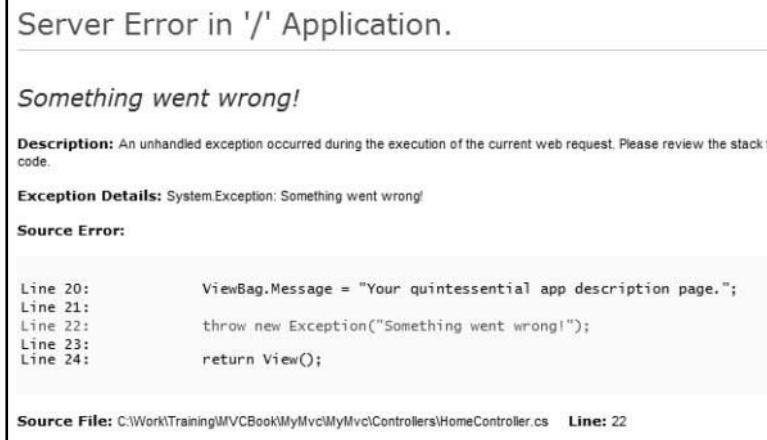


Рис. 16.1. Стандартная страница ошибок ASP.NET

Включение средства специальных ошибок

Первый шаг при обработке ошибок в приложении ASP.NET MVC такой же, как в любом приложении ASP.NET: включение средства *специальных ошибок* ASP.NET.

Это средство поддерживает три режима.

- On. Включает обработку специальных ошибок, отображая страницы специальных ошибок при возникновении различных ошибочных ситуаций.
- Off. Отключает обработку специальных ошибок, отображая стандартную страницу диагностики ошибки при ее возникновении.
- RemoteOnly. Включает обработку специальных ошибок, но только для запросов, которые исходят из удаленной машины. Если вы обращаетесь к приложению из сервера, на котором оно размещено, то увидите стандартную страницу диагностики ошибки, которая помогает отлаживать проблемы с приложением. Однако пользователи будут продолжать иметь дело со страницами специальных ошибок.

Для включения средства специальных ошибок просто измените атрибут mode конфигурационной настройки customErrors внутри элемента system.web в файле web.config на On или RemoteOnly:

```
<customErrors mode="On" defaultRedirect="GenericErrorPage.htm">
    <error statusCode="404" redirect="~/error/notfound"></error>
</customErrors>
```

Имея такую конфигурацию, следующий шаг заключается в расширении стандартного поведения обработки ошибок ASP.NET MVC.

Обработка ошибок в действиях контроллеров

Хотя включение специальной обработки ошибок дает возможность отображать специальную страницу ошибки при возникновении ошибки на сайте, могут быть случаи, когда специальной страницы ошибки оказывается недостаточно.

Для таких случаев ASP.NET MVC предлагает атрибут `HandleErrorAttribute`, который обеспечивает намного более детализированный контроль над тем, что происходит, когда ошибки возникают в действиях контроллеров.

Атрибут `HandleErrorAttribute` открывает два свойства, которые описаны ниже.

- `ExceptionType`. Тип обрабатываемого исключения.
- `View`. Имя представления, которое должно отображаться при возникновении исключения заданного типа.

Просто примените этот атрибут к любому действию контроллера, чтобы сообщить ему, каким образом реагировать при возникновении заданного исключения.

Например, всякий раз, когда возникает исключение базы данных (`System.Data.DataException`) во время выполнения действия `Auction` в показанном ниже примере, ASP.NET MVC будет отображать представление `DatabaseError`:

```
[HandleError(ExceptionType = typeof(System.Data.DataException),
View = "DatabaseError")]
public ActionResult Auction(long id)
{
    var db = new EbuyDataContext();
    return View("Auction", db.Auctions.Single(x => x.Id == id));
}
```

Подобно большинству других атрибутов действия контроллера, `HandleErrorAttribute` может быть также размещен на уровне контроллера, что приводит к его применению ко всем действиям в этом контроллере:

```
[HandleError(ExceptionType = typeof(System.Data.DataException),
View = "DatabaseError")]
public class AuctionsController : Controller
{
    /* Действия контроллера с примененным к ним атрибутом HandleError */
}
```

Определение глобальных обработчиков ошибок

Если необходимо подняться на более высокий уровень, чем даже контроллер, атрибут `HandleErrorAttribute` можно также применить ко всему сайту, регистрируя этот атрибут в качестве глобального обработчика ошибок.

Для регистрации глобального обработчика ошибок откройте файл `/App_Start/FilterConfig.cs` и найдите метод `RegisterGlobalFilters()`. Здесь вы можете видеть, что шаблон ASP.NET MVC уже зарегистрировал глобальный атрибут `HandleErrorAttribute` в `GlobalFilterCollection`.

Чтобы зарегистрировать собственную логику, просто добавьте специальный фильтр в коллекцию глобальных фильтров:

```
public static void RegisterGlobalFilters(GlobalFilterCollection filters)
{
    filters.Add(new HandleErrorAttribute
    {
        ExceptionType = typeof(System.Data.DataException),
        View = "DatabaseError"
    });
    filters.Add(new HandleErrorAttribute());
}
```

Имейте в виду, что по умолчанию глобальные фильтры выполняются в порядке, в котором они зарегистрированы, поэтому обеспечьте регистрацию фильтров ошибок для специфических типов исключений перед любыми другими, более обобщенными фильтрами ошибок (как было показано выше).

В качестве альтернативы методу `filters.Add()` можно предоставить второй параметр, чтобы указать порядок выполнения фильтров:

```
public static void RegisterGlobalFilters(GlobalFilterCollection filters)
{
    filters.Add(new HandleErrorAttribute
    {
        ExceptionType = typeof(System.Data.DataException),
        View = "DatabaseError"
    }, 1);
    filters.Add(new HandleErrorAttribute(), 2);
}
```

Здесь при регистрации фильтров применяются порядковые значения 1 и 2. Это гарантирует, что специальный фильтр `DatabaseError` будет всегда выполняться перед более обобщенным обработчиком ошибок.

Настройка страницы ошибки

Когда обработка специальных ошибок включена, добавленный вами глобальный фильтр `HandleError` перехватывает ошибку и выполняет перенаправление на страницу ошибки (рис. 16.2).

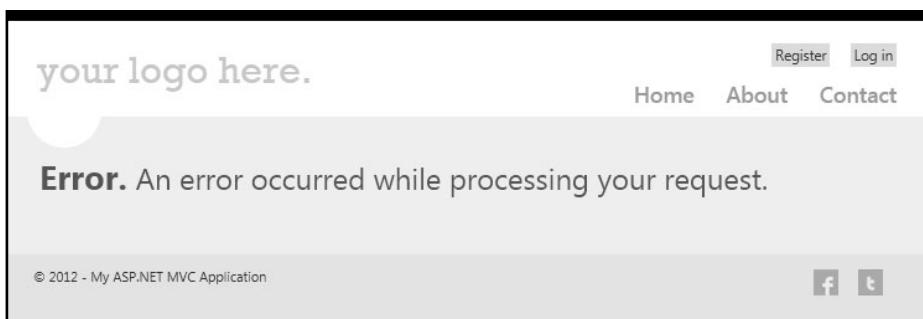


Рис. 16.2. Специальная страница ошибки



Атрибут `HandleErrorAttribute` обрабатывает только 500 ошибок (исключений), генерируемых конвейером ASP.NET MVC; вам понадобится определить специальные правила относительно ошибок для других типов ошибок HTTP, таких как 404.

Если обработка специальных ошибок включена, и вы пользуетесь `HandleErrorAttribute`, исполняющая среда ASP.NET MVC ищет файл `Error.chtml` в папке текущих запросов или в папке разделяемых представлений. При такой настройке `defaultRedirect` (на `GenericErrorPage.htm`) и URI перенаправления для кода состояния игнорируются.

Если обработка специальных ошибок включена, и вы не пользуетесь HandleError Attribute, приложение будет перенаправлять пользователя согласно атрибуту defaultRedirect в файле web.config. Для демонстрации этого поведения за-комментируйте вызов FilterConfig.RegisterGlobalFilters (GlobalFilters.Filters); в Global.asax.cs.

Шаблоны проектов ASP.NET MVC включают стандартную страницу ошибки (~Views/Shared/Error.cshtml), которую можно настроить для своего приложения. Разметка HTML для стандартной страницы ошибки выглядит примерно так:

```
@model System.Web.ASP.NET MVC.HandleErrorInfo  
{  
    ViewBag.Title = "Error";  
}  
  
<hgroup class="title">  
    <h1 class="error">Error.</h1>  
    <h2 class="error">An error occurred while processing your request.</h2>  
</hgroup>
```

Стандартная страница ошибки слишком элементарна, поэтому может возникнуть желание украсить ее немного собственным контентом. Например, может понадобиться предоставить информацию о способе связи пользователей со службой поддержки приложения, в которой помогут решить проблему.

Стандартная страница ошибки – это также строго типизированное представление, которое ссылается на класс модели HandleErrorInfo. Этот класс открывает свойства с информацией об исключении, которое привело к отображению страницы ошибки, а также о действии контроллера, где возникла ошибка.

Регистрация в журнале и трассировка

Когда в приложении возникает проблема, требуется максимально возможный объем сведений для ее прослеживания и устранения причин ее возникновения. И хотя отображение страниц ошибок является неплохим способом информирования пользователей о том, что в приложении произошло исключение, это не позволяет узнать о проблеме *разработчику*.

Чтобы быть информированным обо всех проблемах, возникающих на веб-сайте, к приложению потребуется добавить логику, которая позволит вести учет того, что оно делает, и любых проблем, которые оно испытывает. Такой учет называют *регистрацией в журнале*, и он является, пожалуй, наиболее важным инструментом в арсенале отладки.

Регистрация ошибок в журнале

Когда дело доходит до регистрации исключений в веб-приложении ASP.NET MVC, имеется много вариантов.

Простой вспомогательный класс регистрации в журнале

Рассматриваемые далее примеры ссылаются на специальный вспомогательный класс регистрации по имени Logger, код которого показан ниже. Этот класс регистрирует исключения в журнале событий локальной машины, и если вы хотите опробовать этот код на своей машине, то сначала нужно будет создать источник событий для приложения.

```
public class Logger
{
    public static void LogException(Exception ex)
    {
        EventLog log = new EventLog();
        log.Source = "Ebuy";
        log.WriteEntry(ex.Message);
    }
}
```

Простейший способ сделать это предусматривает использование regedit.exe для добавления нового ключа по имени Ebuy в раздел реестра HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Eventlog\Application.

Простой обработчик try/catch

Первый вариант заключается в помещении блока try/catch внутрь каждого действия контроллера, как показано ниже:

```
public ActionResult About()
{
    try
    {
        ViewBag.Message = "Your quintessential app description page.";
        throw new Exception("Something went wrong!");
    }
    catch (Exception ex)
    {
        LogException(ex);
    }
    return View();
}
```

Тем не менее, этот подход требует добавления множества кода в каждый метод контроллера, и его следует избегать, если только речь не идет о специфичном типе исключения, которое необходимо обрабатывать напрямую, или же существуют специальные требования к регистрации в журнале.

Переопределение Controller.OnException()

Вместо добавления блоков try/catch в каждый метод контроллера можно переопределить метод OnException() контроллера, примерно так:

```
protected override void OnException(ExceptionContext filterContext)
{
    if (filterContext == null)
        base.OnException(filterContext);

    LogException(filterContext.Exception);

    if (filterContext.HttpContext.IsCustomErrorEnabled)
    {
        // Если глобальный фильтр обработки ошибок включен,
        // то в этом нет необходимости.
        filterContext.ExceptionHandled = true;
        this.View("Error").ExecuteResult(this.ControllerContext);
    }
}
```

Даже лучший вариант предусматривает создание базового контроллера, что позволяет иметь код регистрации в журнале только в одном месте. Когда вы переопределяете метод `OnException()`, то должны удостовериться, что переданный контекст не равен `null`, и пометить исключение как обработанное. Если не пометить исключение как обработанное, оно продолжит распространяться вверх по конвейеру ASP.NET MVC.



В случае использования показанного ранее глобального фильтра `HandleError` потребуется удалить код для пометки исключения как обработанного и для отображения представления ошибки, поскольку этот глобальный фильтр уже обработал ошибку.

Специальные фильтры ошибок

Еще один вариант для обработки ошибок в приложении ASP.NET MVC заключается в создании специального фильтра ошибок. Специальные фильтры ошибок позволяют определять логику обработки ошибок и применять ее повсюду на сайте, сокращая объем дублированного кода, который понадобится сопровождать. Специальные фильтры ошибок также позволяют контроллерам оставаться ориентированными на логику обработки запросов, а не на заботу о том, что делать в случае возникновения исключений.

Чтобы создать специальный фильтр ошибок, унаследуйте класс от `HandleErrorAttribute` и переопределите метод `OnException()`. После регистрации исключения в журнале потребуется проверить, включена ли обработка специальных ошибок:

```
public class CustomHandleError : HandleErrorAttribute
{
    public override void OnException(ExceptionContext filterContext)
    {
        if (filterContext == null)
            base.OnException(filterContext);

        LogException(filterContext.Exception);

        if (filterContext.HttpContext.IsCustomErrorEnabled)
        {
            filterContext.ExceptionHandled = true;
            base.OnException(filterContext);
        }
    }

    private void LogException(Exception ex)
    {
        EventLog log = new EventLog();
        log.Source = "Ebuy";
        log.WriteEntry(ex.Message);
    }
}
```

Мониторинг работоспособности ASP.NET

Хотя регистрация в журнале событий является хорошим первым шагом в мониторинге приложения, лучший вариант предусматривает включение мониторинга работоспособности ASP.NET. Мониторинг работоспособности ASP.NET выходит за рамки регистрации исключений в журнале, охватывая отслеживание событий, которые возникают во время жизненного цикла приложения и запроса.

Система мониторинга работоспособности ASP.NET отслеживает перечисленные ниже события:

- события жизненного цикла приложения, включая запуск и останов приложения;
- события безопасности, такие как неудавшиеся попытки входа и запросы авторизации URL;
- ошибки приложения, в том числе необработанные исключения, исключения проверки достоверности запросов, ошибки компиляции и т.д.

Мониторинг работоспособности ASP.NET конфигурируется в разделе `healthMonitoring` файла `web.config` приложения, который содержит три основных подраздела.

- `eventMappings`. Определяет типы событий, которые необходимо отслеживать.
- `providers`. Определяет список доступных поставщиков.
- `rules`. Определяет отображение между событиями и поставщиками, используемыми для регистрации события.

Сконфигурировать мониторинг работоспособности ASP.NET можно следующим образом:

```
<healthMonitoring enabled="true">
  <eventMappings>
    <clear />
    <!-- Регистрировать ВСЕ события ошибок -->
    <add name="All Errors"
      type="System.Web.Management.WebBaseErrorEvent"
      startEventCode="0" endEventCode="2147483647" />
    <!-- Регистрировать события запуска/завершения приложения -->
    <add name="Application Events"
      type="System.Web.Management.WebApplicationLifetimeEvent"
      startEventCode="0" endEventCode="2147483647" />
  </eventMappings>
  <providers>
    <clear />
    <add connectionStringName="DefaultConnection"
      maxEventDetailsLength="1073741823"
      buffer="false" name="SqlWebEventProvider"
      type="System.Web.Management.SqlWebEventProvider" />
  </providers>
  <rules>
    <clear />
    <add name="All Errors Default"
      eventName="All Errors"
      provider="SqlWebEventProvider"
      profile="Default"
      minInstances="1"
      maxLimit="Infinite" minInterval="00:00:00" />
    <add name="Application Events Default"
      eventName="Application Events"
      provider="SqlWebEventProvider"
      profile="Default"
      minInstances="1" maxLimit="Infinite"
      minInterval="00:00:00" />
  </rules>
</healthMonitoring>
```

В готовом виде мониторинг работоспособности ASP.NET включает поставщиков для регистрации в базе данных Microsoft SQL Server и в локальном журнале событий, а также уведомление администраторов по электронной почте. Он также позволяет создавать собственных поставщиков, которые дают возможность регистрации в дополнительных источниках данных.



Чтобы использовать поставщик мониторинга работоспособности для базы данных SQL, понадобится добавить необходимые таблицы в базу данных веб-приложения с помощью команды `aspnet_regsql.exe`, находящейся в каталоге с установленной платформой .NET Framework.

Теперь, когда мониторинг работоспособности включен, нужно обновить ранее созданный специальный фильтр ошибок, чтобы он регистрировал исключения посредством только что настроенных поставщиков.

Поскольку класс `System.Web.Management.WebRequestErrorEvent` системы мониторинга работоспособности не имеет открытых конструкторов, придется сначала создать специальный класс события ошибки веб-запроса:

```
public class CustomWebRequestErrorEvent : WebRequestErrorEvent
{
    public CustomWebRequestErrorEvent(
        string message, object eventSource,
        int errorCode, Exception exception)
        : base(message, eventSource, errorCode, exception)
    {
    }
    public CustomWebRequestErrorEvent(
        string message, object eventSource, int errorCode,
        int eventDetailCode, Exception exception)
        : base( message, eventSource, errorCode,
            eventDetailCode, exception)
    {
    }
}
```

После создания класса следует обновить класс `CustomHandleError` для обращения к специальному классу события ошибки веб-запроса:

```
public class CustomHandleError : HandleErrorAttribute
{
    public override void OnException(ExceptionContext filterContext)
    {
        if (filterContext.HttpContext.IsCustomErrorEnabled)
        {
            base.OnException(filterContext);
            new CustomWebRequestErrorEvent(
                "An unhandled exception has occurred.",
                this, 103005, filterContext.Exception)
                .Raise();
        }
    }
}
```

Имея такой класс и зарегистрировав его в качестве глобального фильтра обработки ошибок, все исключения, возникающие на сайте, будут направляться системе мониторинга работоспособности.

Резюме

При проектировании и построении веб-приложений важно продумать, каким образом будут обрабатываться ошибки, регистрироваться и отслеживаться события, происходящие во время функционирования приложения, и производиться настройка приложения с целью увеличения производительности.

В этой главе вы узнали о мощных встроенных средствах ASP.NET, предназначенных для обработки ошибок, регистрации в журнале и мониторинга работоспособности. Все эти приемы можно использовать для создания надежных веб-приложений с помощью ASP.NET MVC Framework.

Автоматизированное тестирование

В этой книге мы повсеместно поддерживали архитектурные шаблоны и рекомендуемые приемы разработки приложений, в том числе шаблон “модель-представление-контроллер” (MVC), разделение ответственности, SOLID и другие, заявляя о том, что они обеспечивают большую степень повторного использования и возможность сопровождения приложений, в результате повышая их качество. Проблема этих технологий в том, что они направлены на получение преимуществ в долговременной перспективе, что не всегда очевидно и приемлемо в краткосрочных проектах. Например, кого волнует расширяемость компонента, если он никогда не будет расширен на начальной стадии построения приложения?

Истинное значение этих приемов в действительности начинает проясняться на поздних этапах жизненного цикла приложения, когда приложение было выпущено и разработчики должны располагать возможностью исправлять оставшиеся проблемы и добавлять новые средства, одновременно снижая риск введения новых критических изменений в работающую производственную версию приложения.

Тем не менее, упомянутые технологии могут также иметь ценность и в краткосрочной перспективе. К счастью, существует один способ получить эту ценность и одновременно гарантировать постоянный уровень качества приложения: проверка компонентов с помощью приемов автоматизированного тестирования.

В этой главе рассказывается, что нужно для тестирования приложения с использованием разнообразных инструментов и технологий, предназначенных для эффективного написания и выполнения кода и проверки того, что он делает то, для чего был спроектирован. Также будет показано, как применять эти концепции ко всей кодовой базе, с основным акцентом на тестировании приложений ASP.NET MVC на всем пути от контроллеров и служб серверной стороны до кода клиентской стороны, выполняющегося внутри браузера.

Семантика тестирования

Разработка программного обеспечения сосредоточена на создании программных приложений, которые решают задачи путем выполнения любого количества специфичных поведений, которые разработчики называют “требованиями”. Но перед тем как передавать приложения в широкое пользование (т.е. в “производство”), мы должны сначала удостовериться, что эти поведения правильно реализованы. Другими словами, мы должны проверить, что написан высококачественный код, который выполняется в задуманном порядке и является надежным.

Ручное тестирование

Простейший способ проверки, реализована ли функциональная возможность, сводится к запуску приложения и попытке выполнить поведение, как это делал бы обычный пользователь. В рамках настоящей главы мы будем называть этот подход (а также любые другие подходы, предусматривающие проверку с участием человека) *ручным тестированием*; с ним связано множество недостатков.

Люди предрасположены к ошибкам

Прежде всего, ручное тестирование основано на человеческой оценке, а люди, как известно, предрасположены к ошибкам.

Хотя человек, в конечном счете, должен будет выдать окончательную оценку относительно того, корректно ли реализована та или иная функциональная возможность, большинство человеческих оценок вплоть до этого момента были субъективными, иногда до такой степени, что сказать, работает данное средство или нет, становится невозможным.

Например, даже если вы начеку, предупреждены и готовы обнаружить даже самую мелкую ошибку, сможете ли вы найти отличие между строковым значением 1 и целочисленным значением 1, просто видя его на экране? Как известно любому разработчику, который занимался исправлением ошибок какого-нибудь вида, хотя такие проблемы выглядят вполне безобидными, в конечном счете, они могут отражать отличие между работающим приложением и приложением, которое работает отказывается.

Компьютеры являются более эффективными

Следующим возникает вопрос эффективности. Когда пользователь-человек тестирует приложение, он не занимается созданием экземпляров классов и вызовом методов; он взаимодействует с приложением через какой-то пользовательский интерфейс.

Пользователь должен работать с приложением, как было задумано, т.к. изменение приложения с целью упрощения процесса тестирования может поставить под угрозу результаты теста, или же эти изменения могут даже привнести собственные ошибки.

Это часто означает, что единственный тест, воспроизводящий конкретный сценарий, может содержать множество шагов, которые должны выполняться корректно и в нужном порядке, делая ручное тестирование довольно утомительным и трудоемким и еще больше увеличивая шансы допущения ошибок со стороны человека. Компьютеры намного более эффективны при выполнении таких процедур.

Ручное тестирование занимает много времени

И, наконец, самая большая проблема: ручное тестирование занимает много времени — людского времени, которое лучше потратить на что-нибудь другое.

Предположим, например, что разработчик реализует функциональную возможность и должен останавливаться после внесения каждого небольшого изменения, чтобы (корректно) выполнить последовательность потенциально сложных шагов для верификации этого изменения. А теперь подумайте, что произойдет, если это небольшое изменение не заработало, а требуется вносить другое изменение, после чего еще одно, еще одно и т.д.

Далее подумайте о том, что этот же разработчик должен протестировать исключительное условие — т.е. условие, которое очень трудно воспроизвести из-за самой его природы!

Очевидно, что компьютеры намного лучше подходят для выполнения таких типов задач. Таким образом, ответом на все проблемы, присущие ручному тестированию, будет автоматизация этих тестов, чтобы компьютер мог их запускать самостоятельно.

Автоматизированное тестирование

Под *автоматизированным тестированием* понимается идея написания программного обеспечения, которое тестирует другое программное обеспечение и таким образом помогает восполнять недостатки методов ручного тестирования, предлагая компьютеру делать то, что у него получается лучше всего: автоматизировать задачи тестирования. Используя подходы автоматизированного тестирования, люди по-прежнему имеют возможность определять тесты, которые они желают выполнить, и задавать утверждения относительно результатов этих тестов. Самое значительное отличие состоит в том, что после того, как автоматизированный тест впервые создан, его легко запускать столько раз, сколько необходимо, чтобы удостоверяться в его успешном выполнении. Автоматизированные тесты не только проще запускать, чем ручные, но компьютеры способны выполнять те же самые задачи на порядки быстрее, чем люди, радикально сокращая время, необходимое для получения тех же результатов.

Очевидно, что автоматизированное тестирование является именно тем путем, который надо избрать. В оставшихся материалах главы мы рассмотрим разные уровни автоматизированного тестирования, покажем, как создавать проект автоматизированных тестов, и продемонстрируем ряд рекомендуемых приемов для тестирования приложения ASP.NET MVC.

Уровни автоматизированного тестирования

В дополнение к сокращению времени, необходимого на запуск тестов, автоматизированные тесты могут лучше ориентироваться на специфичные компоненты, такие как *тестируемый модуль* (иногда называемый *тестируемой системой*) — термин, относящийся к компоненту, у которого проверяется качество, производительность или надежность.

В контексте разработки программного обеспечения “тестируемый модуль” может относиться к любому уровню программной архитектуры, например, методам, классам, целым приложениям или даже нескольким приложениям, которые работают вместе. Подобно этому неплохо создать множество наборов автоматизированных тестов, причем каждый набор должен быть ориентирован на конкретный уровень программной архитектуры.

В зависимости от целевого архитектурного уровня, каждый набор тестов может быть классифицирован как *модульные тесты*, *интеграционные тесты* или *приемочные тесты* и, как будет показано в последующих разделах, каждая из этих категорий имеет весьма конкретный смысл.

Модульные тесты

Модульные тесты нацелены на проверку самых низких уровней приложения за счет очень узкой направленности. В модульном teste тестируемый модуль — это весьма конкретный, низкоуровневый компонент, такой как класс или даже одиночный метод в классе. На самом деле, поскольку они тестируют настолько специфическую низкоуровневую функциональность, часто для проверки только одного тестируемого модуля требуется несколько модульных тестов.

Цель модульных тестов заключается в проверке действительной логики тестируемого модуля. Короче говоря, не удавшиеся модульные тесты должны служить признаком ошибки в коде и ничего более.

Для достижения такого уровня надежности модульные тесты должны быть *атомарными, повторяемыми, изолированными и быстрыми*. Тесты, которые не удовлетворяют этим четырем фундаментальным требованиям, не могут рассматриваться как настоящие “модульные тесты”.

В последующих разделах приводятся объяснения каждой концепции в контексте автоматизированного тестирования.

Атомарность

Модульный тест должен быть сосредоточен на проверке одной маленькой порции функциональности. Как правило, это будет одиночное поведение или бизнес-сценарий, демонстрируемый классом. Довольно часто модульный тест может быть сужен до одиночного метода в классе (а иногда даже до какого-то условия внутри метода). На практике это эквивалентно коротким тестам с одной лишь парой продуманных и значащих утверждений (`Assert.That([...])`).

Общие ошибки и признаки кода включают:

- десятки строк кода в одном тесте;
- более чем два или три утверждения, особенно когда они применяются к множеству объектов.

Повторяемость

Модульный тест должен выдавать один и тот же результат в любое время в любой среде при условии, что эта среда удовлетворяет известному набору зависимостей (например, .NET Framework). Тесты не могут полагаться на какой-нибудь фактор внешней среды, который не находится под вашим прямым контролем. Например, вы никогда не должны беспокоиться о наличии подключения к локальной сети либо Интернету, доступе к базе данных, правах файловой системы или даже времени суток (имеется в виду `DateTime.Now`).

Общие ошибки и признаки кода включают:

- Тесты успешно проходят при первом выполнении, но некоторые или все отказывают при последующих запусках (или наоборот). Например, вы можете наблюдать комментарий вроде: “Примечание: тест XYZTest должен быть запущен перед этим тестом, иначе он даст отказ!”.

Изолированность/независимость

Как расширение первых двух качеств, модульный тест должен быть полностью изолирован от любой другой системы или теста. Это значит, что модульный тест не должен предполагать, что был выполнен любой другой тест, или зависеть от любой внешней системы (к примеру, базы данных), имеющей специфическое состояние или производящей некоторый результат. Вдобавок, модульный тест не должен создавать или оставлять артефакты, которые могут повлиять на другие тесты. Конечно, это не означает, что модульные тесты не могут разделять между собой методы или даже целие классы – в действительности это даже поощряется. На самом деле это значит, что модульный тест не должен делать предположение, что какой-то другой тест запускался ранее или будет запущен впоследствии. Взамен такие зависимости должны быть

представлены как явные вызовы функций или содержаться в установке тестового при-
способления и методах освобождения, которые выполняются до и непосредственно
после каждого одиночного теста.

Общие ошибки и признаки кода включают:

- доступ в базу данных;
- тесты отказывают, когда подключение к сети или VPN недоступно;
- тесты отказывают, когда не был запущен какой-нибудь внешний сценарий (кро-
ме, возможно, сценария построения);
- тесты отказывают, когда конфигурационные настройки изменяются или некор-
ректны;
- тесты должны выполняться при специфических полномочиях.

Быстрота

Если предположить, что все описанные выше условия удовлетворены, то тесты
должны быть быстрыми (т.е. завершаться за доли секунды). Несмотря на это, по-пре-
жнему полезно явно указывать, что все модульные тесты должны выполняться прак-
тически мгновенно. В конце концов, одним из главных преимуществ комплекта ав-
томатизированных тестов является возможность получить почти мгновенный отклик
о текущем качестве кода. По мере того как время, требуемое для запуска комплекта
тестов, увеличивается, частота его выполнения снижается. Это напрямую транслиру-
ется в более длительный промежуток времени между моментами, когда ошибка была
допущена и когда она была действительно обнаружена.

Общие ошибки и признаки кода включают:

- отдельные тесты выполняются дольше, чем доли секунды.



Внимательные читатели могли заметить, что предшествующий список
могло было бы организовать так, чтобы получился акроним FAIR (fast,
atomic, isolated, repeatable – быстрый, атомарный, изолированный, пов-
торяемый). Хотя это может быть удобно для запоминания четырех клю-
чевых характеристик модульных тестов, порядок их рассмотрения здесь
другой – он отражает степень их важности.

Для демонстрации рассмотренных руководящих принципов ниже приведен при-
мер модульного теста, в котором все они соблюдены:

```
[TestMethod]  
public void CalculatorShouldAddTwoNumbers()  
{  
    var sum = new Calculator().Add(1, 2);  
    Assert.AreEqual(1+2, sum);  
}
```

Этот тест довольно прост и понятен. Сначала он создает новый экземпляр класса
`Calculator` и вызывает его метод `Add()`, передавая два числа, которые должны быть
просуммированы. Затем тест использует метод `Assert.AreEqual()` для определения
утверждения о том, что метод `Add()` делает то, что должен: суммирует два числа.

Мало того, что этот тест легко читать и понимать, он также придерживается всех
указанных ранее руководящих принципов, которые делают его хорошим модульным
тестом.

- Во-первых, он является атомарным: он сосредоточен на проверке поведения метода `Calculator.Add()` (тестируемого модуля) и ни на чем больше. Вдобавок он делает это простым и прямолинейным способом.
- Во-вторых, он является повторяемым: этот тест будет производить тот же самый результат в любое время суток, на любой машине разработчика, независимо от того, сколько раз он выполняется.
- В-третьих, он является изолированным. Этот тест не имеет никаких предусловий, которые должны быть удовлетворены, и не изменяет состояния среды тестирования, когда завершается. Он выполняется независимо от всех других тестов.
- В-четвертых, он является быстрым. Вполне вероятно, что скрытый в методе `Add()` алгоритм может оказаться ужасно неэффективным и отнимать более часа на сложение двух чисел, но, к счастью, это не тот случай. Данный тест выдает надежный результат в пределах нескольких миллисекунд.

Интеграционные тесты

Как противоположность модульным тестам, единственным назначением которых является проверка логики или функциональности специфичного класса или метода, *интеграционные тесты* существуют для проверки взаимодействия (или “интеграции”) между двумя и более компонентами. Другими словами, интеграционные тесты обеспечивают хорошую тренировку для системы, чтобы удостовериться в совместной работе всех отдельных частей с целью достижения желаемого результата: работающего приложения.

Тем не менее, интеграционным тестам присущи недостатки. Поскольку они сосредоточены на верификации совместной работы множества компонентов, становится все труднее изолировать эти компоненты от внешнего мира при попытке их тестирования. Это открывает доступ к целому множеству проблем, начиная с того факта, что обычно нарушается большинство – если не все – описанные ранее правила, которые применяются к модульному тестированию.

Существенными недостатками могут быть низкая скорость и хрупкость интеграционных тестов. Это означает не только то, что они будут выполняться менее часто, чем модульные тесты, но частота *ложно негативных* результатов (отказов тестов, которые не соответствуют нарушенной логике приложения), как правило, оказывается значительно выше.

Когда модульный тест отказывает, то это верный признак ошибки в коде. Напротив, когда не проходит интеграционный тест, это может означать наличие ошибки в коде, но проблема также может быть вызвана другими аспектами в среде тестирования, такими как сбой подключения к базе данных или непредвиденные тестовые данные. Такие *ложно позитивные* результаты, хотя и являются полезным индикатором того, что в среде разработчика что-то пошло не так, обычно только замедляют процесс разработки, отвлекая внимание разработчика от написания работающего кода.

Несмотря на это несколько отрицательное описание, интеграционные тесты являются столь же ценными, как и модульные тесты – если не более ценными.

Предполагая, что преследуется цель избежать потенциальных отвлечений, когда это только возможно, следует также стараться обеспечить широкое покрытие тестами через монолитный комплект модульных тестов, и затем дополнить это покрытие комплектом интеграционных тестов (а не наоборот).

Приемочные тесты

Последним типом теста является *приемочный тест*, который преследует одну цель: удостовериться, что построенная система удовлетворяет запрошенным требованиям. Выражаясь кратко, приемочные тесты проверяют, что система делает все, что пользователи от нее ожидают.

Поскольку приемочные тесты по своему определению обычно довольно субъективны, часто их сложно автоматизировать. Тем не менее, существуют несколько разных технологий, которые позволяют разработчикам автоматизировать выполнение их приложений для проверки того, что они ведут себя так, как должны. За счет применения таких технологий у разработчиков появляется возможность избежать утомительного ручного тестирования своих приложений, сохраняя при этом высокую степень уверенности в том, что приложения будут функционировать так, чтобы пользователи были довольны.

Пользовательское приемочное тестирование

Как узнать, довольны ли пользователи вашим приложением? Нужно просто спросить у них об этом.

Хотя эта глава ориентирована на разработчиков, запускающих свои приложения с целью проверки их правильной работы, подмножество приемочного тестирования под названием *пользовательское приемочное тестирование* (user acceptance testing – UAT) предусматривает вовлечение пользователей непосредственно в процесс разработки за счет передачи им программного обеспечения для тестирования. Даже несмотря на то, что показывать всем пользователям продукт, который, возможно, не обладает качеством финального выпуска, в целом неразумно, пользовательское приемочное тестирование учитывает тот факт, что пользователи приложения являются именно теми, кто действительно может назвать продукт “готовым”.

Более того, участие пользователей в процессе разработки за счет предоставления им возможности использовать частично работающую реализацию приложения может помочь раскрыть как технические, так и коммуникационные проблемы. К тому же на ранних стадиях процесса, когда пользователи способны предоставлять отклики, исправлять эти проблемы гораздо проще и дешевле. Таким образом, хотя пользовательское приемочное тестирование может быть не так легко автоматизировать в стандартное решение, проведение тестирования рано и часто может обеспечить большие преимущества.



Несмотря на то что может найтись кто-то, способный реализовать любые из этих трех типов тестов, первые несколько уровней – модульные и интеграционные тесты – как правило, являются весьма техническими и специфичными для реализации, поэтому они обычно пишутся командой разработчиков для проверки удовлетворения всех технических требований. После этого приложение передается другим группам тестирования (например, команде контроля качества), которые проверяют удовлетворение бизнес-требований.

Что собой представляет проект автоматизированных тестов?

Для создания и выполнения разновидностей автоматизированных тестов, обсуждаемых в настоящей главе, понадобится создать тестовый проект, который будет содержать их.

В мире Visual Studio *тестовый проект* – это относительно обычный проект библиотеки классов, заключающий в себе группу *тестовых классов* (часто называемых *тестовыми приспособлениями*), каждый из которых является нормальным классом .NET, содержащим комплект тестов, представленных в виде методов этого класса. Каждый из таких тестовых методов создает тестируемый модуль, после чего запускает этот компонент для проверки его поведения, используя API-интерфейс тестирования для установки утверждений в отношении компонента (таких как равенство значения свойства ожидаемому значению).



Такой подход, при котором компоненты создаются, запускаются и проверяются (именно в таком порядке), называется шаблоном Arrange-Act-Assert (подготовка, действие, утверждение).

Для того чтобы выполнять автоматизированные тесты в тестовом проекте, этот проект компилируется и передается *средству запуска тестов* – приложению, которое находит все тесты внутри указанной сборки и выполняет их, отслеживая результаты для каждого теста. Пока средство запуска тестов выполняет каждый тест, оно также отслеживает все, что происходит во время прохождения теста, включая любой консольный либо отладочный вывод, который генерирует тест.

После успешного или неудачного выполнения каждого теста средство запуска тестов отображает результаты в сводке, которая позволяет пользователям сразу увидеть, сколько тестов прошло успешно, и сколько завершились неудачей.



В последующих примерах применяются инструменты и API-интерфейсы, по умолчанию входящие во все версии Visual Studio, отличные от Express. Однако если вы работаете с версией Express – или же вас чем-то не устраивают инструменты автоматизированного тестирования, встроенные в Visual Studio и .NET Framework – то знайте, что на выбор доступно несколько других инструментов и инфраструктур автоматизированного тестирования, многие из которых поставляются с открытым кодом. Хотя они могут как обеспечивать, так и не обеспечивать уровень интеграции, аналогичный инструментам Visual Studio, все они стремятся следовать одному и тому же базовому рабочему потоку, поэтому предлагаемый в книге материал должен легко на них транслироваться.

Имеет смысл кратко ознакомиться со всеми доступными инструментами, чтобы определить, какой из них лучше всего подходит для вас и вашей команды. Для начала будет достаточно выполнить поиск в Интернете по фразе “модульное тестирование в .NET”.

Создание тестового проекта в Visual Studio

Создать новый проект модульных тестов в Visual Studio можно несколькими способами. Первый способ предусматривает отметку флашка *Create a unit test project* (Создать проект модульных тестов) в диалоговом окне *New ASP.NET MVC 4 Project*

(Новый проект ASP.NET MVC 4), показанном на рис. 17.1, что приводит к автоматическому созданию проекта модульных тестов и добавлению его в текущее решение веб-сайта ASP.NET MVC.

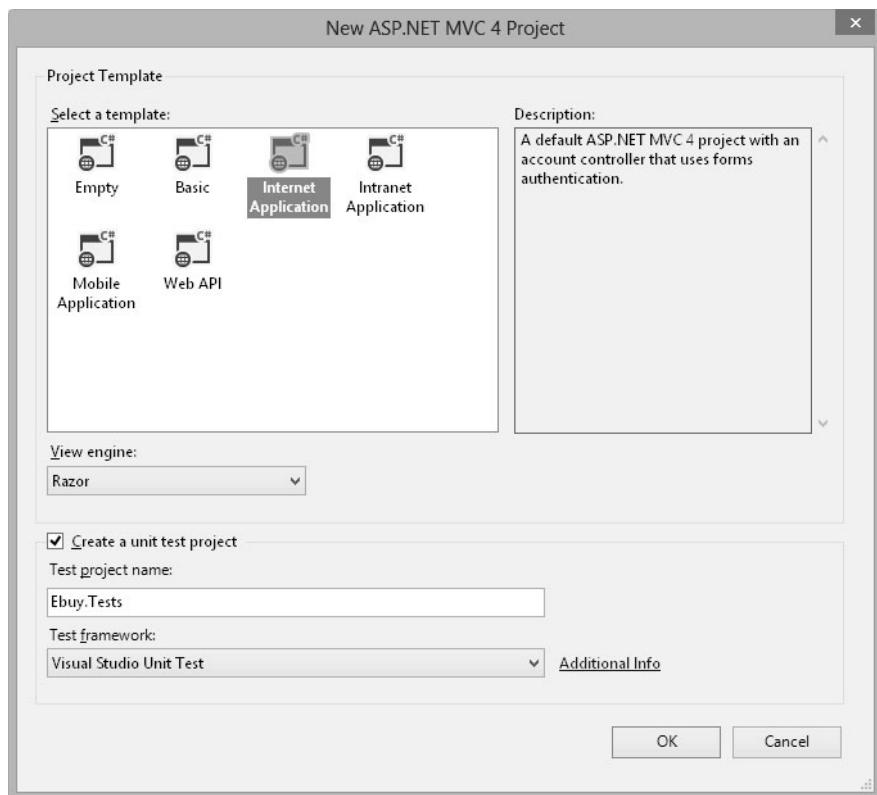


Рис. 17.1. Создание нового проекта модульных тестов в Visual Studio

В качестве альтернативы новый проект модульных тестов в любой момент можно добавить к существующему решению за счет выбора пункта меню **File⇒Add⇒New Project...** (Файл⇒Добавить⇒Новый проект...) и указания типа **Unit Test Project** (Проект модульных тестов) в категории **Test** (Тест) для предпочтаемого языка (рис. 17.2).

Любой из описанных подходов приводит к добавлению в решение нового проекта модульных тестов.

Создание и выполнение модульного теста

Для проверки, все ли работает, попробуйте добавить новый модульный тест, щелкнув правой кнопкой мыши на папке в проекте модульных тестов и выбрав в контекстном меню пункт **Add⇒Unit Test...** (Добавить⇒Модульный тест...). Добавьте в автоматически созданный метод свою логику тестирования и воспользуйтесь вспомогательными функциями API-интерфейса тестирования для установки утверждений в коде.

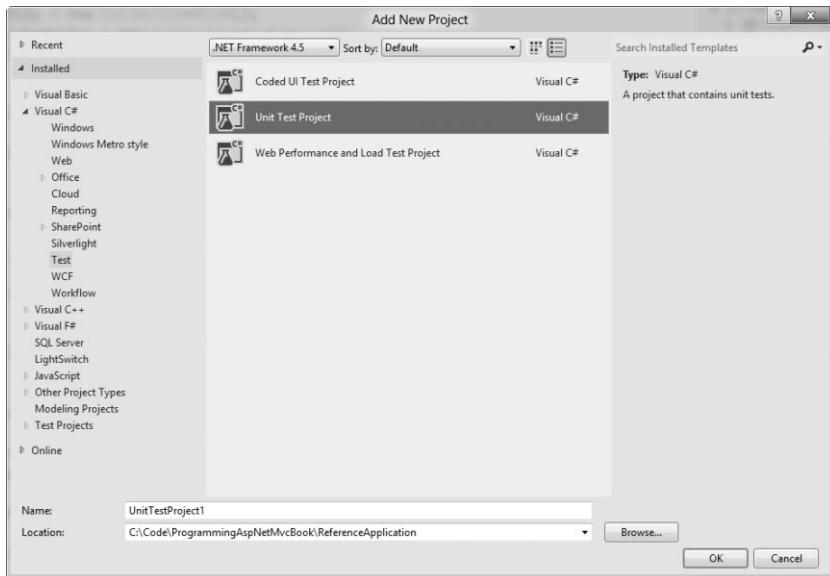


Рис. 17.2. Добавление проекта модульных тестов в существующее приложение

Например, введите следующий код для тестового класса:

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
namespace Ebuy.Tests
{
    [TestClass]
    public class UnitTestExample
    {
        [TestMethod]
        public void CanAddTwoNumbersTogether ()
        {
            var sum = 1 + 2;
            Assert.AreEqual(3, sum);
        }
    }
}
```

По завершении щелкните правой кнопкой мыши где-либо в редакторе кода и выберите в контекстном меню пункт Run Unit Tests... (Запустить модульные тесты...) или нажмите клавиатурное сокращение <Ctrl+R>, <T> в случае стандартных клавиатурных назначений Visual Studio, чтобы запустить тест. Это приведет к компиляции проекта модульных тестов (если это необходимо) и появлению окна Unit Test Explorer (Проводник модульных тестов), показанного на рис. 17.3, для отображения состояния запущенных тестов. Когда тест завершится, вы увидите рядом с ним галочку зеленого цвета, указывающую, что тест был пройден.

Если же тест отказал — скажем, если что-то нарушено в логике, проверяемой тестом (наподобие изменения `var sum = 1 + 2;` на `var sum = 1 + 3;`) — в окне Unit Test Explorer рядом с отказавшим тестом отобразится крестик красного цвета, как показано на рис. 17.4. Вдобавок, если вы щелкнете на отказавшем teste в окне Unit Test Explorer, то сможете просмотреть подробные сведения о том, почему он не прошел.

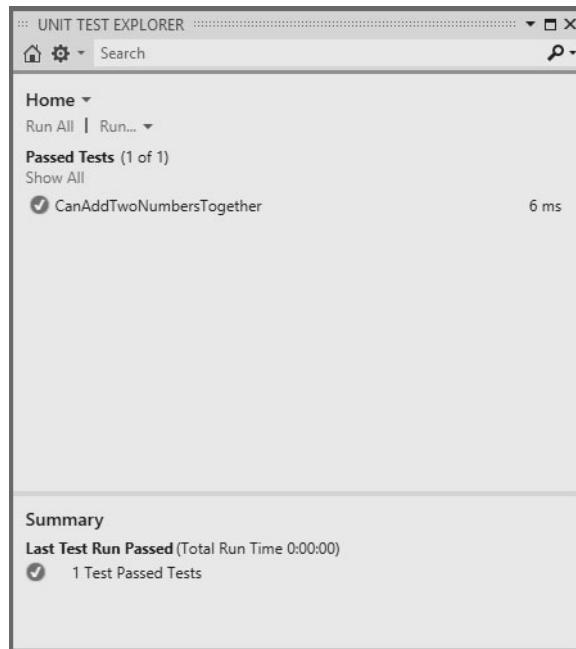


Рис. 17.3. Проверка состояния запущенных тестов

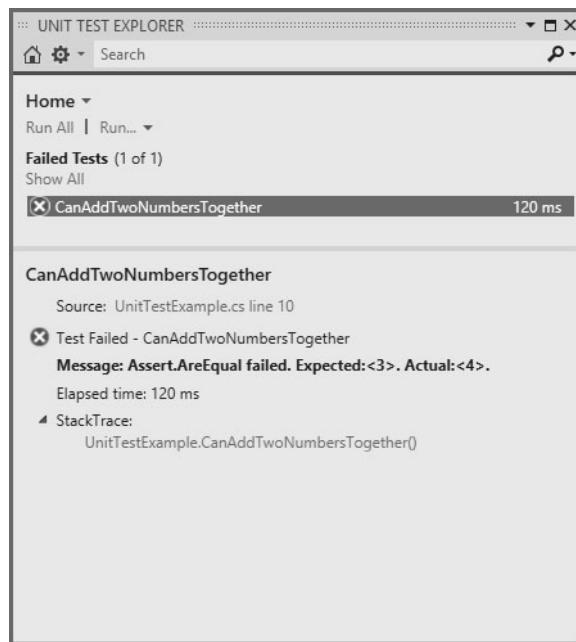


Рис. 17.4. Окно Unit Test Explorer отображает ситуацию, когда тест не прошел, и причины отказа

Обратите внимание в этом примере на сообщение `Message: Assert.AreEqual failed. Expected: <3>. Actual: <4>.` (Сообщение: сбой `Assert.AreEqual`. Ожидалось: `<3>`. В действительности: `<4>`), которое указывает на то, что в вызове вспомогательного метода `Assert.AreEqual()` было задано ожидаемое значение 3, но действительное значение во время выполнения оказалось равным 4, поэтому тест завершился неудачей. Теперь, когда вы знаете, как обращаться с проектом модульных тестов Visual Studio, самое время приступить к тестированию приложения ASP.NET MVC.

Тестирование приложения ASP.NET MVC

Для эффективного тестирования приложения ASP.NET MVC и обеспечения его корректной работы на каждом уровне понадобится комплект автоматизированных тестов, который включает в себя смесь модульных, интеграционных и приемочных тестов.

Хотя существует множество различных подходов для достижения такого покрытия тестами, один из них работает особенно хорошо, используя разделение ответственности ASP.NET MVC и тестируя каждый архитектурный уровень в изоляции. Как будет показано в последующих разделах, тестирование каждого архитектурного уровня самого по себе также удобно тем, что некоторые технологии тестирования лучше работают на одних уровнях, чем на других. Тем не менее, если позаботиться о тщательном тестировании каждого уровня в изоляции, можно иметь намного больше уверенности в том, что эти уровни будут испытывать меньше проблем, когда наступит время тестирования их всех вместе.

Тестирование модели

Поскольку модель является, вероятно, наиболее важной частью приложения, она представляет собой самое логичное место для сосредоточения большинства усилий по тестированию. Как вы увидите при тестировании других уровней, модель также является и простейшим уровнем для тестирования, т.к. она обычно наиболее прямолинейна и имеет меньше всего внешних зависимостей. Другими словами, модели, как правило, состоят из традиционных классов .NET, экземпляры которых легко создавать и выполнять в изоляции.

В целях демонстрации давайте напишем несколько модульных тестов, которые проверяют логику в методе `Auction.PostBid()`:

```
public class Auction
{
    public long Id { get; internal set; }
    public decimal CurrentPrice { get; private set; }
    public ICollection<Bid> Bids { get; private set; }
    // ...

    public Bid PostBid(User user, decimal bidAmount)
    {
        if(bidAmount <= CurrentPrice)
            throw new InvalidBidAmountException(bidAmount, CurrentPrice);
        var bid = new Bid(user, bidAmount);
        Bids.Add(bid);
        CurrentPrice = bidAmount;
        return bid;
    }
}
```

Сосредоточьтесь на позитивных исходах

Перед тестированием любого компонента в первую очередь необходимо воспользоваться моментом и определить простыми словами, что в точности делает этот компонент. Только после этого можно писать модульные тесты для проверки его поведения! Имея данную информацию, необходимо посмотреть, какие тесты понадобится написать для проверки этих поведений, начиная с позитивных исходов, которые ожидаются в наилучшем сценарии (сценариях).

В случае `Auction.PostBid()` метод отвечает за добавление победившего предложения цены в хронологию ценовых предложений для аукционного товара и соответствующее обновление его текущей цены. Другими словами, когда значение победившего предложения цены превышает текущее предложение цены, должно произойти следующее:

- новое предложение цены должно быть добавлено в хронологию ценовых предложений для аукционного товара (`Bids`), содержащую пользователя, который отправил предложение (`user`), и значение предложения (`bidAmount`);
- текущая цена аукционного товара (`CurrentPrice`) должна быть обновлена значением победившего предложения цены (`bidAmount`).

Ниже показано, как эти ожидания транслируются в модульные тесты:

```
using System.Linq;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace Ebuy.Tests
{
    [TestClass]
    public class AuctionTests
    {
        [TestMethod]
        public void ShouldAddWinningBidToBidHistory()
        {
            var user = new User();
            var auction = new Auction { CurrentPrice = 1.00m };
            var bid = auction.PostBid(user, 2.00m);

            CollectionAssert.Contains(auction.Bids.ToArray(), bid);
        }

        [TestMethod]
        public void ShouldUpdateCurrentPriceWithWinningBidAmount()
        {
            var user = new User();
            var auction = new Auction { CurrentPrice = 1.00m };
            var bid = auction.PostBid(user, 2.00m);

            Assert.AreEqual(auction.CurrentPrice, 2.00m);
        }
    }
}
```

Скомпилировав и запустив эти тесты с применением средства запуска тестов Visual Studio, вы должны увидеть, что они оба успешно прошли, указывая на то, что метод `Auction.PostBid()` делает именно то, что предполагалось (рис. 17.5).

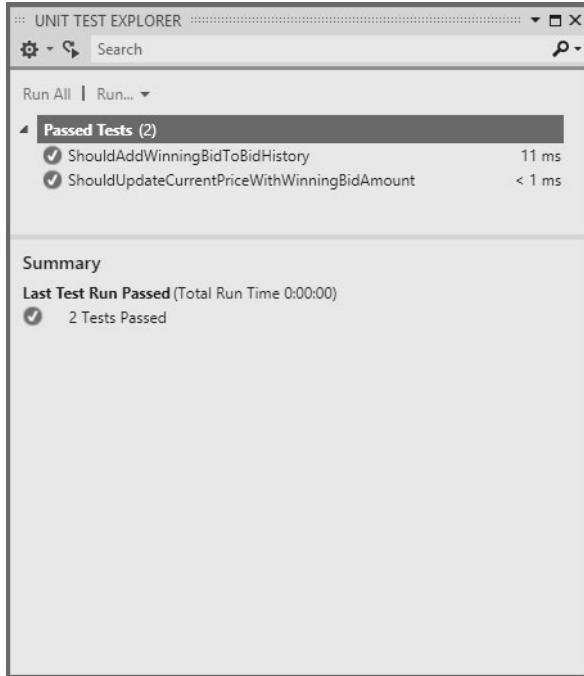


Рис. 17.5. Проверка того, что метод Auction.PostBid() делает то, что должен

Защититесь от негативных исходов

Теперь, когда выполнена проверка того, что метод `Auction.PostBid()` делает то, что *предполагалось*, необходимо удостовериться, что он *не делает* того, что *не должен*. Другими словами, давайте нарушим его работу!

Начнем с переключения ожидаемого условия на противоположное. Предыдущие тесты проверяли корректность поведения, когда значение победившего предложения цены превышало текущее предложение цены, поэтому теперь мы будем проверять, что должно предположительно произойти, когда значение несостоявшегося предложения цены *меньше* текущего предложения цены:

```
[TestMethod]
[ExpectedException(typeof(InvalidBidAmountException))]
public void ShouldThrowExceptionWhenBidAmountIsLessThanCurrentBidAmount()
{
    var user = new User();
    var auction = new Auction { CurrentPrice = 1.00m };
    auction.PostBid(user, 0.50m);
    // Никаких утверждений, потому что предыдущая строка генерирует исключение!
}
```

Этот тест пытается отправить предложение цены (`0.50m`), которое меньше текущего победившего предложения цены (`1.00m`), и использует атрибут `ExpectedExceptionAttribute` для указания на то, что логика внутри теста должна генерировать исключение типа `InvalidBidAmountException`. Если тест завершает выполнение без генерации такого исключения, тест считается не пройденным.

Что может произойти в случае передачи пользователя, равного `null`? Метод `Auction.PostBid()` в настоящее время не проверяет пользователя на предмет равенства `null`, но должен делать это. Здесь мы видим, что проблему в коде можно обнаружить, просто применения *процесс* модульного тестирования, а не разрабатывая для этого какой-то модульный тест.

Разработка через тестирование

Вместо немедленной корректировки кода `Auction.PostBid()` давайте воспользуемся подвернувшейся возможностью и опробуем стиль написания автоматизированных тестов, который называется *разработкой через тестирование* (*test-driven development* – TDD) или иногда *разработкой в стиле “сначала тест”*.

Разработка через тестирование следует мантре “красный-зеленый-рефакторинг”, которая означает, что вы начинаете с не прошедшего теста (“красный свет”), затем пишете абсолютно минимальный (и вполне может быть, что самый запутанный и неуклюжий) код, который делает возможным прохождение этого теста (“зеленый свет”). И, наконец, получив прошедший тест для гарантии, что код работает, необходимо вернуться и провести рефакторинг только что написанного кода с целью его очистки и приведения к стандартам, принятым в проекте.

Для применения подхода TDD к методу `Auction.PostBid()` потребуется определить цель – защититься от значений `null` для пользователей – и начать в “красном” состоянии с не прошедшим тестом:

```
[TestMethod]
[ExpectedException(typeof(ArgumentNullException))]
public void ShouldThrowAnExceptionWhenUserIsNull()
{
    var auction = new Auction { CurrentPrice = 1.00m };
    auction.PostBid(null, auction.CurrentPrice + 1);
    // Никаких утверждений, потому что предыдущая строка генерирует исключение!
}
```

Этот тест создает новый аукционный товар и пытается передать значение `null` в качестве параметра `user` методу `PostBid()`. Данный тест также использует атрибут `ExpectedExceptionAttribute` для утверждения, что код внутри теста генерирует исключение, которое мы ожидаем, почти как в предыдущем примере.

Разница между тем, что делалось в предшествующем примере, и тем, что делается сейчас, заключается в том, что если запустить этот тест с текущей реализацией метода `Auction.PostBid()`, то он не пройдет, т.к. метод пока еще не содержит проверки параметра `user` на предмет `null`.

Теперь мы находимся в состоянии “красный” с не прошедшим тестом, поэтому давайте обеспечим прохождение теста. Для этого просто добавим условие, которое проверяет значение `user` на равенство `null`:

```
public Bid PostBid(User user, decimal bidAmount)
{
    if(user == null)
        throw new ArgumentNullException("user");
    if(bidAmount <= CurrentPrice)
        throw new InvalidBidAmountException(bidAmount, CurrentPrice);
    var bid = new Bid(user, bidAmount);
    Bids.Add(bid);
    CurrentPrice = bidAmount;
    return bid;
}
```

Имея проверку значения `user` на равенство `null`, наш новый модульный тест будет проходить успешно: мы теперь находимся в “зеленом” состоянии. Как правило, после достижения “зеленого” состояния со всеми прошедшими тестами можно уделить некоторое время на пересмотр только что написанного кода, чтобы попробовать сделать его более чистым, быстрым либо улучшить еще в каком-нибудь отношении.

Написание чистых автоматизированных тестов

Несмотря на то что они пишутся для совершенно другой цели, не имеющей отношения к “производственному” коду, автоматизированные тесты все равно являются кодом, поэтому к ним применяется большинство стандартных приемов кодирования.

Дублированный код

Для начала дублированный код не рекомендуется в автоматизированных тестах в не меньшей степени, чем в производственном коде. Рассмотрим для примера тесты, которые только что были написаны. Почти все они начинаются со следующих двух строк:

```
var user = new User();
var auction = new Auction { CurrentPrice = 1.00m };
```

Хотя эти две строки создают объекты, которые критически важны для выполнения тестов, они не особенно относятся к тестируемой логике. Например, написанные до сих пор тесты не заботились о точном значении свойства `CurrentPrice` аукционного товара, а только о том, что оно имеется.

К счастью, почти все инфраструктуры модульного тестирования предоставляют способ для выполнения кода установки непосредственно перед выполнением каждого теста. API-интерфейс тестирования в Visual Studio поддерживает это, позволяя использовать атрибут `TestInitializeAttribute` для указания метода в тестовом классе, который содержит код установки теста.

В примере 17.1 показано, как выглядит класс модульного теста после применения атрибута `TestInitializeAttribute` для централизованного размещения всего дублированного кода.

Пример 17.1. AuctionTests.cs

```
using System;
using System.Linq;
using Microsoft.VisualStudio.TestTools.UnitTesting;
namespace Ebuy.Tests
{
    [TestClass]
    public class AuctionTests
    {
        private User _user;
        private Auction _auction;
        [TestInitialize]
        public void TestInitialize()
        {
            _user = new User();
            _auction = new Auction { CurrentPrice = 1.00m };
        }
        [TestMethod]
```

```
public void ShouldAddWinningBidToBidHistory()
{
    var bid = _auction.PostBid(_user, _auction.CurrentPrice + 1);
    CollectionAssert.Contains(_auction.Bids.ToArray(), bid);
}

[TestMethod]
public void ShouldUpdateCurrentPriceWithWinningBidAmount()
{
    var winningBidAmount = _auction.CurrentPrice + 1;
    _auction.PostBid(_user, winningBidAmount);
    Assert.AreEqual(_auction.CurrentPrice, winningBidAmount);
}

[TestMethod]
[ExpectedException(typeof(InvalidBidAmountException))]
public void ShouldThrowExceptionWhenBidAmountIsLessThanCurrentBidAmount()
{
    _auction.PostBid(_user, 0.50m);
}

[TestMethod]
[ExpectedException(typeof(ArgumentNullException))]
public void ShouldThrowAnExceptionWhenUserIsNull()
{
    _auction.PostBid(null, _auction.CurrentPrice + 1);
}
}
```

Как видите, перемещение дублированной логики инициализации в метод `TestInitialize()` делает каждый отдельный тест намного проще и более относящимся к делу.

Именование

Поскольку код автоматизированного теста никогда не потребляется приложением (отличным от средства запуска тестов), существуют несколько заметных исключений из стандартных правил кодирования. Одной из областей, в которых возникают такие исключения, является именование. Имена классов и методов чрезвычайно важны в производственном коде, поскольку они информируют разработчиков о том, как класс или метод принимает участие в приложении — что делает метод или за что отвечает класс.

Именование не менее важно и в контексте автоматизированного тестирования, но с одним фундаментальным отличием: тестовые классы всегда служат единственной цели действовать в качестве контейнеров для тестовых методов, а тестовые методы всегда предназначены для тестирования конкретного модуля.

Следовательно, тестовые классы должны называться для модулей, на которые нацелены их тестовые методы, и тестовые методы должны описывать проверяемое ими поведение. Взгляните еще раз на имена классов и методов, используемые в тестах, которые были показаны в этой главе ранее. До сих пор мы тестировали только один целевой класс, поэтому имеется единственный тестовый класс: ему назначено имя `AuctionTests`, указывая на то, что все методы в этом классе нацелены на класс `Auction` внутри модели.

Имена тестовых методов гораздо более интересны. Обратите внимание, что тестовые методы имеют не просто очень описательные имена, но эти имена выглядят почти как действительные фразы. Например, тест ShouldAddWinningBidToBidHistory осуществляет проверку, что класс Auction должен добавлять победившее предложение цены в свою хронологию предложений. Имена тестовых методов могут быть настолько длинными, насколько допускает язык (к примеру, ShouldThrowExceptionWhenBidAmountIsLessThanCurrentBidAmount).

При выборе схемы именования допускается определенная свобода, объясняемая ролью, которую исполняют тестовые классы и методы: они предназначены для проверки кода и оповещения о наличии потенциальных проблем. Таким образом, если тест не пройден, помогает уже то, что само имя является максимально описательным. В этом случае очень легко определить, какая часть логики приложения отказалась.

Тестирование контроллеров

Самое замечательное в инфраструктуре ASP.NET MVC Framework то, что она написана с учетом тестируемости. Это означает, что очень просто создать экземпляр практически любого класса в инфраструктуре и выполнить его внутри модульного теста, в точности как это можно делать с моделями.

Контроллеры ASP.NET MVC не являются исключением из этого правила: в действительности они представляют собой просто классы, а действия контроллеров – это всего лишь методы. Взгляните для примера на стандартный контроллер HomeController:

```
using System.Web.Mvc;
namespace Ebuy.Website.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            ViewBag.Message = "Your app description page.";
            return View();
        }
        public ActionResult About()
        {
            ViewBag.Message = "Your quintessential app description page.";
            return View();
        }
        public ActionResult Contact()
        {
            ViewBag.Message = "Your quintessential contact page.";
            return View();
        }
    }
}
```

Чтобы протестировать действия контроллера HomeController, нужно только создать новый экземпляр с использованием стандартного конструктора и вызвать действие. В целях демонстрации давайте выполним простой тест, который проверяет то, что действие Index возвращает представление:

```
[TestClass]
public class HomeControllerTests
{
    [TestMethod]
    public void ShouldReturnView()
    {
        var controller = new HomeController();
        var result = controller.Index();

        Assert.IsInstanceOfType(result, typeof(ViewResult));
    }
}
```

Обратите внимание, что модульный тест имеет возможность создать экземпляр `HomeController` и вызвать действие полностью за пределами конвейера ASP.NET MVC. Здесь можно отметить мощь слабого связывания ASP.NET MVC Framework и понять, почему действия контроллера возвращают объекты `ActionResult`, а не выполняют оставшуюся часть запроса. Это не только обеспечивает большую мощь и гибкость во время выполнения запросов, но также позволяет модульным тестам проверять поведение каждой части запроса по отдельности.

Тестирование логики доступа к данным

После примера с действием `Index` контроллера `HomeController` давайте перейдем к примеру, в котором предусмотрен доступ к данным. Взглянем на действие `Auction` в исходной реализации `AuctionsController` из начала книги, до его переделки в главе 8 для использования более тестируемого шаблона `Repository`:

```
public ActionResult Auction(long id)
{
    var db = new EbuyDataContext();
    var auction = db.Auctions.Find(id);

    return View(auction);
}
```

Как и в случае тестов, которые были написаны ранее, мы начнем процесс тестирования с определения того, что в точности это действие контроллера по плану должно делать, на нормальном языке. Оно извлекает аукционный товар с заданным идентификатором из базы данных и отображает его в представлении. Просто, не так ли?

Но когда наступает время написания модульного теста, возникают некоторые сложности, подобные перечисленным ниже.

- Должны ли в базе данных существовать хоть какие-то аукционные товары для извлечения тестом?
- Если аукционные товары существуют, какие идентификаторы должен использовать тест для их извлечения?
- Предполагая, что аукционные товары существуют, и мы знаем их идентификаторы, как тест может проверить, что был извлечен корректный аукционный товар? (Например, если тест запрашивает аукционный товар с идентификатором 5, как удостовериться, что он не получил аукционный товар с идентификатором 8?)

Простым ответом на все эти вопросы является обеспечение того, чтобы тест создавал аукционный товар в базе данных, гарантируя наличие аукционного товара для извлечения при каждом своем запуске.

Итак, давайте опробуем следующий код и посмотрим, как он работает:

```
using Ebuy.Website.Controllers;
using Microsoft.VisualStudio.TestTools.UnitTesting;
namespace Ebuy.Tests.Website.Controllers
{
    [TestClass]
    public class AuctionsControllerTests
    {
        private Auction _auction;
        [TestInitialize]
        public void TestInitialize()
        {
            using(var db = new EbuyDataContext())
            {
                _auction = new Auction { Title = "Test Auction" };
                db.Auctions.Add(_auction);
                db.SaveChanges();
            }
        }
        [TestMethod]
        public void ShouldRetrieveAuctionById()
        {
            var controller = new AuctionsController();
            dynamic result = controller.Auction(_auction.Id);
            Assert.AreEqual(_auction.Id, result.Model.Id);
            Assert.AreEqual(_auction.Title, result.Model.Title);
        }
    }
}
```

Обратите внимание на то, как этот тестовый класс применяет метод `TestInitialize()` для добавления тестового аукционного товара в базу данных перед выполнением логики теста. Затем тест может ссылаться на значение `Id` тестового аукционного товара для его извлечения из базы данных. Наконец, для подтверждения того, что действие контроллера извлекло корректный аукционный товар, мы обеспечиваем одинаковые значения для свойств `Id` и `Title`.

Следует отметить, что перед тем, как можно будет успешно запустить этот тест, потребуется настроить строку подключения Entity Framework в конфигурации тестового проекта. Чтобы сконфигурировать строку подключения (если это еще не сделано), добавьте в `App.config` следующий раздел:

```
<connectionStrings>
<add name="DefaultConnection"
      connectionString="Data Source=(LocalDb)\v11.0;Initial Catalog=EBuy.Tests;
      Integrated Security=true"
      providerName="System.Data.SqlClient" />
</connectionStrings>
```

Когда все необходимое сконфигурировано и готово к работе, запустите тест, чтобы удостовериться в его прохождении и должной работе контроллера. Вы только что использовали автоматизированное тестирование для проверки логики вашего контроллера!

Рефакторинг модульных тестов

Хотя тесты в предыдущем примере могут проверять логику в `AuctionsController`, в то же самое время они также непреднамеренно тестируют логику доступа к данным. Несмотря на ценность этого подхода в том, что он делает неплохую работу по имитации взаимодействия, которое будет происходить в производственной среде, он также страдает от всех нежелательных побочных эффектов, привносимых интеграционными тестами, которые делают такие виды тестов медленными и ненадежными. Самая большая проблема с одновременным тестированием двух компонентов возникает, когда тесты не проходят. Как быстро определить, какой компонент в этом виноват?

Лучший способ локализовать проблемы, когда они возникли — сосредоточиться на проверке логики каждого компонента с индивидуальным нацеливанием модульных тестов, а затем поместить несколько компонентов вместе, чтобы посмотреть, насколько хорошо они интегрируются. Поскольку вы тестируете каждый компонент в приложении, перестаньте принимать во внимание, в чем точно заключается работа этого компонента, и удостоверьтесь, что тесты нацелены только на поведение этого компонента, не распространяясь случайно на более широкую область.



При написании автоматизированных тестов — особенно модульных тестов — гораздо проще сосредоточиться на тестируемом модуле, если предполагается, что все остальные компоненты, с которыми тестируемый модуль взаимодействует, работают должным образом. На самом деле, когда вы пишете тесты для всех компонентов подобным способом, то получаете в конечном итоге всесторонний комплект весьма целенаправленных тестов! Имея такой всесторонний комплект автоматизированных тестов, в целом намного легче определить проблемные компоненты, когда что-то пошло не так, поскольку тесты, ориентированные на данный компонент, и будут теми, которые нарушены, в отличие от группы тестов, предназначенных для других, несвязанных компонентов.

Например, когда вы пишете тесты для действия контроллера, которое извлекает данные из репозитория и выполняет с ними какую-то работу, в действительности вы тестируете логику внутри действия контроллера, а *не* уровень доступа к данным, извлекающий эти данные. Проблема заключается в том, что тесты, которые были только что написаны, проверяют оба уровня. В следующем разделе мы рассмотрим способ обхода этой проблемы.

Имитация зависимостей

Когда вы сталкиваетесь с проблемами вроде только что описанной, такие технологии, как рассмотренный в главе 8 шаблон *Repository*, начинают демонстрировать свою ценность. Шаблоны наподобие указанного вводят уровень абстракции, который позволяет легко заменять производственные компоненты фиктивными компонентами, поэтому появляется возможность контролировать данные, предоставляемые тестируемому модулю (в этом случае действию контроллера).

Замена компонентов их фиктивными аналогами для целей тестирования называется *имитацией*, а сами заменяющие компоненты — *тестовыми дубликатами*. На них также ссылаются как на *фиктивные объекты*, *объекты-заглушки* и (чаще всего) *пробные объекты*.

Поскольку при использовании таких технологий имеется возможность гарантировать, что поведение зависимости никогда не будет изменяться, можно приступать к

превращению интеграционных тестов в модульные тесты, удалив внешнюю зависимость и сосредоточившись только на логике контроллера. Для демонстрации выполнения такого преобразования давайте посмотрим, как можно применить технологию имитации к контроллеру `AuctionsController` после его рефакторинга с учетом шаблона `Repository` в главе 8:

```
using System.Web.Mvc;
namespace Ebuy.Website.Controllers
{
    public class AuctionsController : Controller
    {
        private readonly IRepository _repository;
        public AuctionsController()
            : this(new DataContextRepository(new EbuyDataContext()))
        {
        }
        public AuctionsController(IRepository repository)
        {
            _repository = repository;
        }
        public ActionResult Index()
        {
            var auctions = _repository.Query<Auction>();
            return View(auctions);
        }
        public ActionResult Auction(long id)
        {
            var auction = _repository.Single<Auction>(id);
            return View(auction);
        }
    }
}
```

Обратите внимание на наличие двух конструкторов: один из них принимает `IRepository`, используемый контроллером для всего доступа к данным, а другой создает стандартную реализацию `IRepository`, которая будет применяться в производственной среде.

В этом случае, если контроллер создан с помощью стандартного конструктора (т.е. конструктора, используемого ASP.NET MVC Framework), он будет применяться в производственной реализации `IRepository` (`DataContextRepository`). С другой стороны, конструктор, принимающий `IRepository`, обеспечивает место, в которое модульные тесты могут внедрять пробный объект для управления данными, предоставляемыми контроллером.

Ручное создание пробных объектов

Прежде чем можно будет воспользоваться тем фактом, что контроллер `AuctionsController` способен принимать пробный объект `IRepository`, потребуется создать его. Пожалуй, наиболее простым способом создания пробного объекта `IRepository` является его самостоятельное написание, т.е. создание класса, который реализует интерфейс `IRepository` и существует для единственной цели – содействие автоматизированным тестам.

В следующем фрагменте кода показан пример очень простой реализации IRepository, которая позволяет управлять тем, что возвращает метод Single<TModel>():

```
public class MockAuctionRepository : IRepository
{
    private readonly Auction _auction;
    public MockAuctionRepository(Auction auction)
    {
        _auction = auction;
    }
    public TModel Single<TModel>(object id) where TModel : class
    {
        return _auction as TModel;
    }
    public IQueryable<TModel> Query<TModel>() where TModel : class
    {
        throw new System.NotImplementedException();
    }
}
```

Класс MockAuctionRepository имеет несколько важных отличительных признаков.

- Во-первых — и это важнее всего — этот класс открывает методы, которые ведут себя одинаково при каждом их вызове, что делает взаимодействие с данным классом со стороны других компонентов весьма предсказуемым.
- Во-вторых, пробный класс позволяет тестам управлять данными, возвращаемыми методом Single<TModel>(). Он делает это путем возвращения объекта Auction, который передается конструктору каждый раз, когда выполняется метод Single<TModel>().
- В-третьих, класс MockAuctionRepository реализует абсолютный минимум контракта IRepository, который необходим для автоматизированных тестов. Другими словами, MockAuctionRepository нацелен на специфичный тестовый сценарий и не предназначен быть репозиторием общего назначения. Это проявляется в том, что метод Query<TModel>() генерирует исключение NotImplementedException, и он делает это намеренно, т.к. нам известно, что тестируемый модуль (метод AuctionsController.Auction()) должен только вызвать метод IRepository.Single<TModel>(). Если в ходе наших тестов метод Query<TModel>() когда-либо выполнится, генерируется указанное исключение, а тест (вполне корректно) не пройдет.

Теперь давайте перепишем ранее показанный (в разделе “Тестирование логики доступа к данным”) интеграционный тест для применения нового пробного класса репозитория:

```
[TestClass]
public class AuctionsControllerTests
{
    [TestMethod]
    public void ShouldRetrieveAuctionById()
    {
        var auction = new Auction { Id = 123 };
        var mockRepository = new MockAuctionRepository(expectedAuction);
```

```
        var controller = new AuctionsController(mockRepository);
        dynamic result = controller.Auction(expectedAuction.Id);
        Assert.AreSame(expectedAuction, result.Model);
    }
}
```

Здесь видно, что тест использует основанную на репозитории реализацию `AuctionsController`, создавая и передавая экземпляр нового класса `MockAuction Repository`, чтобы занять место стандартной реализации `IRepository`, основанной на базе данных. Затем тест может вызвать метод `controller.Auction()` с ожиданием, что тот вызовет метод `MockAuctionRepository.Single<Auction>()`. После этого в тесте устанавливаются утверждения относительно объекта `result.Model`, необходимые для доказательства, что это тот же самый экземпляр `Auction (expectedAuction)`, который тест предоставил пробному репозиторию.



Обратите внимание, что этот тест устанавливает значение свойства `Auction.Id`, хотя метод установки для этого свойства помечен модификатором доступа `internal` и класс находится в другой сборке, отличной от сборки с классом `Auction` (т.е. не в том же самом контексте `internal`). Это возможно благодаря применению к проекту `Ebuy.Core` (где определен класс `Auction`) показанного ниже атрибута `InternalsVisibleTo` уровня сборки, который открывает тестовому проекту `Ebuy.Tests` все внутреннее устройство `Ebuy.Core`:

```
[assembly: InternalsVisibleTo("Ebuy.Tests")]
```

Такой подход предлагает еще одну выгоду: так как тест заменяет основанную на базе данных реализацию `IRepository` новой пробной реализацией `IRepository`, больше нет необходимости в обеспечении того, чтобы в базе данных существовал экземпляр `Auction` со специфическим `Id` (тест создает этот `Auction` самостоятельно).

Точно так же имеется возможность замены внешней зависимости базы данных пробным объектом и перевода интеграционного теста в модульный.

Использование имитирующей инфраструктуры

Несмотря на то что созданные вручную пробные классы, такие как `MockAuction Repository`, являются великолепным способом замены производственных компонентов во время тестирования, с ними связан ряд проблем.

Первым делом, невзирая на то, что пробные классы используются только в процессе тестирования, они по-прежнему являются действительными реализациями интерфейсов приложения, а это означает не только необходимость написания разработчиками большего объема кода для построения эффективных модульных тестов, но также обязательное их сопровождение по мере развития кодовой базы. Указанная проблема усугубляется тем фактом, что разработчики обычно пишут множество пробных реализаций единственного интерфейса для различных целевых использований этого интерфейса, в частности как в примере `MockAuctionRepository` реализован только метод `Single<Auction>()` и ничего более. Подумайте об эффекте от изменения лежащего в основе интерфейса, если существуют десятки пробных реализаций этого интерфейса!

К счастью, доступна альтернатива ручному созданию пробных классов: применение для этого *имитирующей инфраструктуры*.

Имитирующая инфраструктура — это инфраструктура, которая предоставляет разработчикам API-интерфейс для динамического создания пробных классов на лету и простого конфигурирования этих классов под конкретную ситуацию. Короче говоря, имитирующие инфраструктуры предлагают все преимущества вручную созданных пробных классов с долей выполненной работы, как во время первоначальной разработки тестов, так и в период сопровождения комплекта модульных тестов.



Несмотря на широкое разнообразие имитирующих инфраструктур, доступных разработчикам .NET, все они выполняют по существу одну и ту же задачу: создание пробных объектов на лету. Так как их основные отличия сводятся к форме синтаксиса, применяемого для создания пробных объектов, на выбор конкретной имитирующей инфраструктуры обычно влияют персональные предпочтения разработчика. В этой книге мы решили демонстрировать примеры с использованием инфраструктуры с открытым кодом Moq (<http://code.google.com/p/moq/>), но вы можете избрать любую другую инфраструктуру. Например, существует немало инфраструктур с открытым кодом, среди которых Rhino Mocks (<http://hibernatingrhinos.com/open-source/rhino-mocks>), EasyMock.NET (<http://sourceforge.net/projects/easymocknet/>), Nmock (<http://www.nmock.org/>) и FakeItEasy (<https://github.com/FakeItEasy/FakeItEasy>). Кроме того, доступен ряд коммерческих имитирующих инфраструктур.

Для примера взгляните на следующий фрагмент, в котором созданный вручную класс `MockAuctionRepository` из предыдущего примера заменен пробным объектом `IRepository`, генерируемым инфраструктурой Moq:

```
[TestClass]
public class AuctionsControllerTests
{
    [TestMethod]
    public void ShouldRetrieveAuctionById()
    {
        var expectedAuction = new Auction { Id = 123 };
        var mockRepository = new Moq.Mock<IRepository>();
        mockRepository
            .Setup(repo => repo.Single<Auction>(expectedAuction.Id))
            .Returns(expectedAuction);
        var controller = new AuctionsController(mockRepository.Object);
        dynamic result = controller.Auction(expectedAuction.Id);
        Assert.AreEqual(expectedAuction, result.Model);
    }
}
```

Здесь видно, что имитирующая инфраструктура способна заместить интерфейс `IRepository` с помощью всего нескольких строк кода.

Сначала тест динамически создает новый пробный экземпляр `IRepository` через объект `Moq.Mock<T>: newMoq.Mock<IRepository>()`. Затем тест сообщает пробному объекту, как он должен себя вести, используя метод `Setup()` имитирующей инфраструктуры Moq для указания вызова, который пробный объект должен ожидать (`repo.Single<Auction>(expectedAuction.Id)`), и метод `Returns()` для указания значения, которое должно возвращаться в ответ на ожидаемый вызов.

Конечный результат этого примера заключается в том, что мы имеем возможность заменить вручную созданный класс `MockAuctionRepository` динамической пробной реализацией `IRepository`, которая предоставляет все преимущества пробного класса, но требует намного меньшего кода и последующего сопровождения. И поскольку имитирующая инфраструктура существенно сокращает объем усилий, необходимых для эффективной имитации зависимостей, этот подход помогает разработчикам писать более производительные тестовые комплекты, которые содержат намного больше модульных, чем интеграционных тестов.

Тестирование представлений

Обеспечив успешное тестирование уровня моделей и контроллеров, самое время перейти к тестированию представлений. Однако есть и плохие новости: это очевидно трудная задача.

Чтобы понять, почему ожидается насколько много работы, задайте себе тот же вопрос, который мы задавали повсеместно в этой главе перед тем, как начинать тестирование чего-либо: что здесь в точности планируется тестировать? Подумайте о взаимодействии пользователя с приложением ASP.NET MVC. Инфраструктура ASP.NET MVC Framework обеспечивает визуализацию HTML-разметки для браузера пользователя, но поскольку здесь участвует браузер, то это только начало!

Как обсуждалось в главе 4, назначение современной веб-страницы выходит далеко за рамки простой визуализации HTML-разметки и CSS-стилей в браузере. Расширенные технологии JavaScript и AJAX развили нынешнюю веб-сеть от простого механизма доставки контента до многонаправленного взаимодействия, превращающего браузер в полноценную платформу разработки приложений внутри себя — и эта платформа может иметь очень мало общего с ASP.NET MVC!

Все это сводится к тому, что тестирование уровня представлений приложения является гораздо более сложным процессом, чем просто создание какого-то объекта и написание нескольких строк кода для проверки его поведения. На самом деле “уровень представлений” сам может быть фактически разделен на множество разных уровней: визуализированная HTML-разметка, которая содержит изначальный контент и структуру страницы, код JavaScript, содержащий логику приложения, и правила CSS, которые стилизуют все это. Все три указанных аспекта страницы должны быть безупречными, чтобы достичь желаемого опыта взаимодействия — и все они требуют очень разных подходов и технологий тестирования.

Тестирование логики приложения в браузере

Хотя приведенные выше примеры показывают, что основанное на тексте автоматизированное тестирование веб-сайта безусловно возможно, преимущества текстового подхода могут быть весьма ограничены, поскольку контент, который браузер получает, и способ, который он выбирает для визуализации этого контента, могут оказаться двумя очень разными вещами. И это становится еще хуже, когда принять во внимание, насколько отличающимися методами разнообразные браузеры могут визуализировать одну и ту же HTML-разметку. Для того чтобы действительно проверить, каким образом сайт будет взаимодействовать с реальными пользователями, его неизбежно придется загрузить в браузер и посмотреть, как сайт будет выглядеть для пользователей.

Такой подход называется *тестированием в браузере*, потому что он предусматривает открытие действительного браузера и применение виртуальных щелчков кнопками

мыши и нажатий клавиш на клавиатуре для имитации действий, которые будут совершать пользователи сайта. Очевидно, что простейший способ выполнения тестирования в браузере заключается в ручном открытии браузера и взаимодействии с сайтом, как это делал обычный пользователь. Тем не менее, эта глава посвящена автоматизированному тестированию, поэтому мы сосредоточим внимание на разнообразных способах поручить выполнение этих задач компьютеру — другими словами, на организации *автоматизированного тестирования в браузере*.

Существует целый ряд инструментов, которые позволяют выполнять тестирование в браузере изнутри комплекта автоматизированных тестов, но в этой главе мы собираемся рассмотреть инструмент под названием WatiN (<http://watin.org/>).

Чтобы приступить к использованию WatiN в своем решении, понадобится с помощью диспетчера пакетов NuGet добавить в список ссылок тестового проекта пакет WatiN. После добавления упомянутой ссылки необходимо создать нормальный тестовый класс и тестовый метод — как это делалось в предшествующих разделах — и импортировать пространство имен WatiN.Core. Затем понадобится применить экземпляр класса WatiN.Core.IE для открытия экземпляра Internet Explorer и начать взаимодействие с браузером внутри тестов.

Например, следующий тест использует Internet Explorer для перехода на веб-сайт Ebuy:

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using WatiN.Core;

namespace Ebuy.Tests.Website.Browser
{
    [TestClass]
    public class AuctionTests
    {
        [TestMethod]
        public void ShouldNavigateToAnAuctionListingFromTheAuctionsList()
        {
            const string baseUrl = "http://localhost:65193";
            using (var browser = new IE(baseUrl + "/auctions", true))
            {
                var auctionDiv = browser.Div(Find.ByClass("auction"));
                var auctionTitle = auctionDiv.Element(Find.ByClass("title")).Text;
                auctionDiv.Links.First().Click();
                Assert.IsFalse(string.IsNullOrWhiteSpace(auctionTitle));
                Assert.AreEqual(
                    auctionTitle,
                    browser.Element(Find.BySelector("h2.title")).Text);
            }
        }
    }
}
```

Этот тест инструктирует Internet Explorer относительно перехода на `http://localhost:65193/auctions` — URL для отображения списка аукционных товаров, выполняющийся на локальном веб-сервере разработки с портом 65193. После загрузки страницы тест находит первый элемент `Auction` (`<div class="auction">`), затем просматривает его для обнаружения названия аукционного товара (``) и сохраняет это название с целью подтверждения того, что загружена корректная страница, позже в teste.

Как только такое обнаружение завершено, наступает время выполнения самого теста. Тест находит первую ссылку в элементе `Auction` и инициирует щелчок, используя метод `.Click()` ссылки, который перенаправляет браузер на страницу подробностей выбранного аукционного товара.

Наконец, тест проверяет, что все работает ожидаемым образом, устанавливая утверждение относительно то, что название аукционного товара (которое извлечено из страницы списка аукционных товаров) совпадает с текстом в элементе `<h2 class="title">` на странице подробностей. Если все идет, как запланировано, проблем с нахождением элемента `<h2>` возникать не должно, а он должен содержать надлежащее название аукционного товара.



Если вы пытаетесь выполнить модульный тест и получаете сообщение о том, что средство запуска тестов не может загрузить сборку `Interop.SHDocVw`, это означает, что объект взаимодействия с СОМ не был обнаружен.

Чтобы исправить эту проблему, найдите сборку `Interop.SHDocVw` в списке ссылок тестового проекта и измените значение свойства `Embed Interop Types` (Встраивать типы взаимодействия) на `false`, а значение свойства `Copy Local` (Копировать локально) – на `true`.

Это должно позволить средству запуска тестов взаимодействовать с `Internet Explorer` и успешно выполнять тесты.

Несмотря на простоту, этот пример является мощной демонстрацией возможностей автоматизированного тестирования в браузере. Обратите внимание на разнообразные способы, которыми тесты могут взаимодействовать с браузером через развитый API-интерфейс `WatIN`. К примеру, класс `IE` обеспечивает представление браузера, которое тесты могут запрашивать повсеместно, в то время как вспомогательные методы, доступные в классе `Find`, предлагают различные способы быстрого и эффективного поиска элементов в DOM. Можно не только находить и анализировать элементы браузера; разнообразные методы, доступные на элементах браузера, такие как метод `.Click()`, предоставляют тестам способ имитации взаимодействия пользователя с браузером.

Покрытие кода

Покрытие кода – это прием анализа кода, который оценивает качество проверки приложения комплектом автоматизированных тестов на основе того, сколько строк кода выполняется во время прогона этого комплекта. Покрытие кода обычно выражается в процентах; например, если компонент содержит 100 строк кода, а комплект тестов приводит к выполнению 75 из них, то говорят, что компонент имеет покрытие кода 75%.

Для оценки покрытия кода в проекте потребуется воспользоваться соответствующим инструментом. Этот инструмент выполняет комплект автоматизированных тестов внутри процесса профилировщика и отслеживает каждую строку кода, которая выполняется во время прогона каждого модульного теста. По завершении запуска теста инструмент анализа покрытия кода обычно создает отчет, показывающий, насколько хорошо тесты покрывают различные части приложения.

Если вы работаете в версии `Visual Studio`, отличной от Express, то вам повезло, т.к. в IDE-среде имеется встроенный инструмент анализа покрытия кода. Для его использования выберите пункт меню `Unit Test`⇒`Analyze Code Coverage` (Модульный

тест⇒Анализировать покрытие кода) и затем вариант Selected Tests (Выбранные тесты) или All Tests (Все тесты).

Это приведет к выполнению инструментом ваших модульных тестов, анализу полученных результатов покрытия кода и отображению их в окне Code Coverage Results (Результаты покрытия кода) внутри IDE-среды (как показано на рис. 17.6).

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
ebuy.core.dll	26	36.62 %	45	63.38 %
{ } Ebuy	26	36.62 %	45	63.38 %
Auction	0	0.00 %	24	100.00 %
Auction()	0	0.00 %	9	100.00 %
PostBid(Ebuy.User, decimal)	0	0.00 %	15	100.00 %
Bid	0	0.00 %	6	100.00 %
Bid(Ebuy.User, decimal)	0	0.00 %	6	100.00 %
DataContextRepository	3	33.33 %	6	66.67 %
DataContextRepository(System.Data.Entity.DbContext)	0	0.00 %	2	100.00 %
Query<T>()	3	100.00 %	0	0.00 %
Single<T>(object)	0	0.00 %	4	100.00 %
EbuyDataContext	0	0.00 %	4	100.00 %
EbuyDataContext()	0	0.00 %	4	100.00 %
EbuyDataContextInitializer	23	100.00 %	0	0.00 %
Seed(Ebuy.EbuyDataContext)	23	100.00 %	0	0.00 %
InvalidBidAmountException	0	0.00 %	5	100.00 %
InvalidBidAmountException(decimal, decimal)	0	0.00 %	5	100.00 %

Рис. 17.6. Результаты покрытия кода, полученные с помощью Visual Studio

Как видно на этом рисунке, Visual Studio может показать, вплоть до уровня метода, сколько кода “покрыто” или “не покрыто” вашим комплектом автоматизированных тестов.

В рассматриваемом случае покрыто 63,38% кода всей сборки `ebuy.core.dll`. Если вы предполагали, что данный проект имел всесторонний комплект автоматизированных тестов, то это число может оказаться сюрпризом, т.к. вы могли ожидать что-то близкое к 100%.

Чтобы увидеть, что препятствует стопроцентному покрытию кода, сначала попробуем найти прорехи в покрытии. Быстрый взгляд на список методов позволяет выяснить, что два метода вообще не покрыты: `DataContextRepository.Query<T>()` и `EbuyDataContext.Initializer.Seed()`.

Эти методы имеют одну общую характеристику: они оба взаимодействуют с базой данных. Кроме того, `EbuyDataContext.Initializer.Seed()` выполняется только при первом создании базы данных и никогда после этого.

Такая информация влечет за собой несколько последствий, но одной из наиболее важных моментов, на которые она указывает, является то, что наши тесты всегда выполняются в отношении существующей базы данных, но никогда не создают новую базу. Это может соответствовать проекту, и тогда отсутствие покрытия кода в этом методе вполне приемлемо. Или же, может выясниться, что мы сделали очень важную ошибку и наши тесты должны создавать новую базу данных при каждом прогоне!

Не менее интересен тот факт, что метод `DataContextRepository.Query<T>()` никогда не выполнялся. Эта информация может также указывать, что нам надо написать дополнительные тесты, нацеленные на данный метод, или, возможно, такой метод вообще не нужен.

В конечном счете, это все оценочные вызовы, которые должны быть сделаны. В зависимости от ситуации, могут существовать весьма веские причины оставить части кода не покрытыми автоматизированными тестами и, таким образом, не получить “стопроцентное покрытие кода” – и это нормально!

Миф о стопроцентном покрытии кода

Оценка покрытия кода комплектом тестов является отличным способом удостовериться, что вы работаете с максимально возможной частью приложения, и вы должны стремиться тестировать столько строк кода, сколько реально получится. Однако хотя и целесообразно пытаться протестировать каждую строку кода в приложении, с поиском возможности стопроцентного покрытия кода связаны две проблемы.

1. *Это практически невозможно.* Несмотря на наличие множества способов разработки кода, который очень легко тестировать (как показано в следующем разделе), существует много ситуаций, очень сложных для покрытия автоматизированными тестами. Примером такой ситуации является любой компонент, взаимодействующий напрямую с `HttpRequest` – компонентом инфраструктуры, который требует немалой работы для своего создания. Простейший и наиболее эффективный путь для тестирования этих компонентов предусматривает запуск их внутри конвейера ASP.NET, что нарушает большинство руководящих принципов модульного тестирования, упомянутых ранее в этой главе.
2. *Это дает ложное чувство безопасности.* Рассмотрим пример модульного теста, который был разработан в начале главы. Предположим, что этот тест обеспечивает стопроцентное покрытие кода метода `Calculate.Add()` – т.е. он выполняет каждую строку кода в этом методе без сбоев. Означает ли это, что метод `Calculate.Add()` никогда не может отказать? Мы его протестирували только со значениями 1 и 2; значит ли это, что метод должно образом поддерживает любое заданное значение? (Подсказка: попробуйте вызов `Calculate.Add(double.MaxValue, double.MaxValue)`.)

Все это вовсе не говорит о том, что не следует стремиться к максимально достижимому проценту покрытия кода. Просто имейте указанные моменты в виду, и не считайте свой комплект тестов неудачным, если не получается достигнуть мистического стопроцентного покрытия кода!

Разработка поддающегося тестированию кода

Если эффективное тестирование программного обеспечения сосредоточено на проверке того, что компонент или приложение будут работать ожидаемым образом в производственной среде, то все факторы, мешающие этому коду выполняться максимально близко к тому, как он будет выполняться в производственной среде, могут сделать код более трудным в проверке, т.е. менее “поддающимся тестированию”.

Например, вспомните самый последний компонент, с которым вы имели дело, и задайте себе несколько вопросов.

- Сколько действий по установке и конфигурированию требуется для подготовки компонента к тестированию?
- Со сколькими другими компонентами (которые могут требовать собственного дополнительного конфигурирования) взаимодействует этот компонент?
- Зависит ли компонент от любых библиотек третьих сторон, баз данных, внешних веб-служб или даже локальной файловой системы?
- Насколько надежными и согласованными являются эти зависимости? Например, есть ли у вас тестовая веб-служба, которая всегда возвращает те же самые результаты?

Все эти моменты могут влиять на способность к тестиированию программного компонента. Возьмем, к примеру, два метода, показанные ниже. Хотя оба метода содержат только одну строку кода, первый из них является намного более тестируемым, чем второй метод, который представляет один из наиболее сложных сценариев для надежного тестиирования. Вот первый метод `GetVersionNumber()`:

```
public static int GetVersionNumber()
{
    return 1;
}
```

Тестируемый метод `GetVersionNumber()` делает тот факт, что он всегда возвращает один и тот же результат (значение 1 типа `int`), независимо от того, сколько раз он вызывается. Возможно, даже более важно то, насколько доступным является этот метод. Так как он имеет уровень доступа `public`, любой компонент может выполнять его и просматривать его ответ. Модификатор `static` также означает, что потребители не должны создавать экземпляр содержащего его класса, чтобы вызвать метод – используется просто вызов метода.

Метод `CallRemoteWebService()`, с другой стороны, представляет собой совершенно другую историю:

```
private ServiceResult CallRemoteWebService()
{
    return new WebService("http://thirdparty.com/service")
        .GetServiceResult("some special value");
}
```

Для начала, этот метод закрыт от внешнего мира посредством уровня доступа `private`. Это означает, что для его выполнения тест должен сначала вызвать другой метод в том же классе, вводя несвязанную логику, чего мы стремимся избегать. Этот также означает, что тест должен будет создать экземпляр класса, чтобы получить возможность вызвать любой из его методов, что по причине существования зависимости класса может оказаться нетривиальной задачей.

Следующий аспект, связанный с `CallRemoteWebService()`, заключается в том, что этот метод обращается к внешней зависимости (удаленной веб-службе), которая привносит целый ряд проблем, возникающих при попытке выполнения этого метода. Активна ли веб-служба? Могут ли возникнуть проблемы с сетью при попытке доступа к веб-службе? Возвращают ли она корректные данные? Если это служба третьей стороны, понадобится ли позже оплачивать счет за пользование ею?

И последняя, наиболее заметная проблема, которая затрудняет тестиирование метода `CallRemoteWebService()`, связана с тем, как он обращается к зависимости в виде внешней веб-службы. Обратите внимание, что метод создает новый экземпляр класса `WebService` и применяет жестко закодированный URL для ссылки на внешнюю службу. По этой причине не существует способа протестировать этот метод без выполнения действительного обращения к внешней производственной веб-службе, а это значит, что нет возможности управлять поведением зависимости в виде веб-службы.



Будьте осторожны с ключевым словом `new` – обычно оно указывает на возможность применения приема внедрения зависимостей, помогающего создавать более слабое связывание между компонентами. Когда компонент получает экземпляр другого объекта через внедрение зависимостей, это поведение намного легче заменять тестовым объектом в процессе тестирования. Но если компонент создает экземпляр самостоятельно, замена такой логики практически невозможна.

Следовательно, любой тест, который выполняет этот метод, в действительности является тестом интеграции с внешней веб-службой. Хотя это не так уж плохо, это означает, что данный метод никогда не будет подвергаться *модульному* тестированию, т.к. это нарушит все вышеупомянутые руководящие принципы для настоящего модульного теста.

Если принять во внимание еще и последствия описанной ситуации, то все окажется намного хуже. Применять модульное тестирование невозможно не только к этому методу, но и к другим методам, которые *вызывают* этот метод. При довольно большом числе таких методов в приложении очень скоро станет невозможно провести модульное тестирование всего приложения!

Резюме

Автоматизированное тестирование – это отличный способ обеспечения постоянного качества и функциональности целого приложения. Традиционно применять технологии автоматизированного тестирования к веб-приложениям ASP.NET было нелегко. Тем не менее, слабо связанная архитектура, которую предоставляет ASP.NET MVC Framework, делает автоматизированное тестирование приложений ASP.NET MVC довольно простым.

Представленная в главе связка ASP.NET MVC Framework с полезными шаблонами, приемами и инструментами достаточно скоро позволит обрести уверенность, обеспечиваемую полным комплектом автоматизированных тестов.

Автоматизация построения

Для того чтобы перейти от низкоуровневого исходного кода к полноценно функционирующему приложению, должно быть проделано немало вещей. Например, приложения, написанные на статическом языке вроде C#, понадобится скомпилировать и, возможно, скопировать в специальную папку (такую как папка `bin` в случае веб-приложения ASP.NET). Приложение может также требовать для корректного функционирования несколько других артефактов, подобных изображениям, файлам сценариев и даже схемам целых баз данных. Процесс подготовки таких артефактов, а также любых других компонентов, требуемых для функционирования приложения, часто называют “построением”.

В предыдущих главах было показано, что в то время как людям довольно трудно выполнять повторяющиеся задачи без погрешностей, компьютеры обрабатывают такие задачи легко и максимально точно. В главе 17 демонстрировался отличный пример этого, когда компьютеру поручалось проведение автоматизированного тестирования приложения.

В этой главе тема автоматизации разнообразных аспектов, связанных с разработкой программного обеспечения, расширяется за счет применения приемов автоматизации к построению и развертыванию приложений. Помимо этого мы покажем, как использовать автоматизацию для улучшения взаимодействия между членами команды разработки и другими группами,участвующими в создании, проверке и доставке программного обеспечения.



Хотя примеры в этой главе демонстрируют распространенные сценарии построения и развертывания с применением инструментов MSBuild и Team Foundation Server производства Microsoft, существуют многочисленные альтернативы, как коммерческие, так и с открытым кодом, и каждая из них поддерживает собственный способ создания и выполнения сценариев построения.

Однако, несмотря на их различия, почти все инфраструктуры построения и развертывания управляются теми же самыми фундаментальными концепциями и технологиями. Таким образом, следуя примерам в этой главе, имейте в виду, что важны только *концепции* автоматизированного построения и развертывания, а не то, какие инструменты используются для их реализации.

Создание сценариев построения

Перед тем, как можно будет начать автоматизацию построения, сначала потребуется определить, что конкретно должен делать компьютер. Для этого служат сценарии построения – файлы, которые содержат набор задач, предназначенных для выполнения компьютером.

Действительный формат и синтаксис сценариев построения варьируется в зависимости от применяемого инструмента. Тем не менее, независимо от того, как они написаны, сценарии построения могут содержать логику, которая распространяется далеко за пределы простой компиляции кода, и включает такие задачи, как выполнение комплекта модульных тестов приложения, оценка качества исходного кода приложения, генерация и запуск сценариев базы данных – словом, почти любую задачу, которую только можно вообразить.

Среды разработки вроде Visual Studio часто закладывают основу для автоматизации построения через концепции наподобие “решений” и “проектов”, которые позволяют очень легко определить артефакты, требуемые приложению, а также то, что делать с этими артефактами для получения работающего приложения. К примеру, файл проекта C# в Visual Studio (.csproj) может включать коллекцию файлов исходного кода C# и логику запуска компилятора C# (csc.exe) для компиляции этих файлов исходного кода в сборку либо исполняемый модуль .NET. Независимо от того, насколько полезными они могут быть внутри IDE-среды Visual Studio, проекты и решения только слегка затрагивают то, что возможно делать в мире автоматизированного построения.

Проекты Visual Studio являются сценариями построения

Хотя существуют многочисленные инструменты для написания сценариев, помогающие создавать автоматизированные задачи, вас может удивить, что в вашем расположении уже имеется очень мощный инструмент написания сценариев, установленный как часть Visual Studio: *Microsoft Build Engine* (или, как его более часто называют, *MSBuild*). Средство MSBuild полагается на сценарии, определенные с использованием специальной схемы XML для выполнения разнообразных задач. На самом деле все файлы проектов и решений Visual Studio представляют собой файлы MSBuild со специальными расширениями, и каждый раз, когда вы нажимаете <F5>, среда Visual Studio передает эти файлы MSBuild для компиляции ваших приложений!

Добавление простой задачи построения

Чтобы удостовериться в сказанном, откройте проводник Windows, перейдите в папку EBuy.Website внутри папки решения Ebuy и откройте файл EBuy.Website.csproj в текстовом редакторе, щелкнув на этом файле правой кнопкой мыши, выбрав в контекстном меню пункт Open With (Открыть с помощью) и указав предпочтительный текстовый редактор. Внутри файла проекта вы найдете XML-узел <Project> с различными дочерними элементами, такими как <Import>, <PropertyGroup> и <ItemGroup> – все эти элементы работают вместе для сообщения MSBuild о том, как построить проект.

Переместитесь в конец файла проекта и найдите следующие закомментированные строки:

```
<!-- Чтобы модифицировать процесс построения, добавьте свою задачу внутрь  
одной из целей ниже и удалите с нее комментарий. Существуют и другие  
похожие точки расширения; см. Microsoft.Common.targets.
```

```
<Target Name="BeforeBuild">
</Target>
<Target Name="AfterBuild">
</Target> -->
```

Как должно быть понятно по именам, эти две цели — `BeforeBuild` и `AfterBuild` — позволяют выполнять задачи до и после выполнения остальных задач построения.

Чтобы посмотреть, насколько легко модифицировать файл построения, давайте изменим его для отображения сообщения по завершении процесса построения. Для этого удалите комментарий с цели `AfterBuild` и поместите в нее вызов MSBuild-задачи `<Message>`:

```
<!-- Чтобы модифицировать процесс построения, добавьте свою задачу внутрь
одной из целей ниже и удалите с нее комментарий. Существуют и другие
похожие точки расширения; см. Microsoft.Common.targets.
<Target Name="BeforeBuild">
</Target>
-->
<Target Name="AfterBuild">
    <Message Importance="High" Text="**** The build has completed! ****" />
</Target>
```

Выполнение построения

После внесения такого изменения наступило время для выполнения построения, чтобы увидеть его в действии. Здесь доступны два варианта: построение внутри Visual Studio и запуск MSBuild напрямую в командной строке.

Построение внутри Visual Studio

Выполнение MSBuild изнутри Visual Studio осуществляется легко — на самом деле, это в точности то, что вы делали, используя Visual Studio для разработки и выполнения приложений .NET. Откройте решение, содержащее проект, и нажмите одно из многих клавиатурных сокращений Visual Studio (такое как `<Ctrl+B>`), которое запускает процесс построения.

После того как построение завершено, вы увидите ваше специальное сообщение, отображаемое в окне вывода Visual Studio. Например:

```
3>---- Rebuild All started: Project: Ebuy.Website, Configuration: Debug Any CPU ----
3>  Ebuy.Website -> C:\Code\EBuy\trunk\Website\bin\Ebuy.Website.dll
3>  **** The build has completed! ****
===== Rebuild All: 3 succeeded, 0 failed, 0 skipped =====
```

Построение в командной строке

С проектами и решениями Visual Studio связан один интересный момент: для их построения открывать их в Visual Studio не обязательно. Вместо этого можно выполнить “сценарий построения” проекта, просто запустив MSBuild напрямую в командной строке.

Чтобы сделать это, найдите в меню `Start` (Пуск) и откройте окно командной строки Visual Studio. В результате запустится процессор командной строки Windows (`cmd.exe`) с автоматическим конфигурированием среды, обеспечивающим доступ к инструментам .NET, включая MSBuild.

После этого можно запустить команду `msbuild` и передать ей имя файла решения или проекта в качестве аргумента для выполнения построения:

```
msbuild Ebuy.sln
```

Эта команда запускает средства MSBuild для указанного решения и выдает немало текста по сравнению с окном вывода Visual Studio. Где-то внутри этого многословного текста вы должны заметить заданное специальное сообщение AfterBuild:

```
[...]
AfterBuild:
The build has completed!
Done Building Project "C:\Code\EBuy\Website\Ebuy.Website.csproj" (default targets).
[...]
```

Возможности безграничны!

Хотя добавление специального сообщения в вывод построения Visual Studio может показаться довольно впечатляющим, имейте в виду, что это всего лишь простой пример. В действительности этот пример демонстрирует саму возможность вставки специальной логики в процесс построения, позволяя выполнять буквально любую логику, которую можно вызывать из кода или сценария .NET посредством командной строки.

Автоматизация процесса построения

Несмотря на удобство запуска сценария построения за счет нажатия клавиатурной комбинации в Visual Studio или выполнения в командной строке, эти подходы далеко не идеальны, т.к. требуют вмешательства человека. Настоящую ценность при создании сценариев построения представляет собой *автоматизация построения*, при которой компьютер способен выполнять сценарии построения без привлечения человека.

После создания сценариев построения, которые детализируют то, что должно происходить во время автоматизированного построения, следующий шаг в устраниении человеческого вмешательства заключается в передаче сценария *службе автоматизации построения*, более известной как *сервер построения*. Сервер построения – это служба, которая функционирует постоянно, ожидая возможности выполнить предоставленные ей сценарии построения.

Одним из продуктов серверов построения, которым предпочитают пользоваться команды разработки .NET, является Team Foundation Server (TFS) от Microsoft. Продукт Team Foundation Server – это популярный выбор, потому что он интегрирует несколько важных концепций жизненного цикла разработки приложений, среди которых управление исходным кодом, отслеживание единиц работы, построение отчетов, а также автоматизированное построение и развертывание.



Так как эта глава сосредоточена только на демонстрации концепций автоматизированного построения, мы предполагаем, что вы уже установили и сконфигурировали Team Foundation Server и применяете его в качестве системы управления исходным кодом. Если вы хотите отслеживать приведенные далее примеры, но не имеете доступа к установке Team Foundation Server, то можете избрать вариант бесплатного или недорогого доступа к хостингу Team Foundation Server (<http://tfspreview.com>), который предоставляет все средства, рассматриваемые в этой главе. Если же вы решили пользоваться одним из множества других популярных серверов автоматизации построения, то применять изложенные в главе концепции к этому инструменту придется самостоятельно.

Типы автоматизированных построений

Поскольку сценарии построения могут запускать практически любую логику, какую только можно вообразить, важно определить для каждого построения контекст, который укажет, что конкретно построение рассчитывает выполнить.

Например, в следующем списке представлено несколько распространенных типов автоматизированных построений.

- **Непрерывные построения.** Непрерывные построения инициируются всякий раз, когда любой член команды фиксирует изменение в кодовой базе, и их основное назначение в том, чтобы предоставить почти немедленную обратную связь относительно качества зафиксированного изменения. Для того чтобы проверить качество, выполняемые непрерывным построением задачи обычно ограничены компиляцией приложения и запуском комплекта модульных тестов, который обеспечивает базовую проверку работоспособности кода. Эти модульные тесты завершаются за короткий промежуток времени. Из-за частоты инициирования непрерывных построений очень важно, чтобы эти типы построений завершались как можно быстрее, чтобы ускорить отклик и избежать накопления множества построений друг над другом.
- **Последовательные построения.** Последовательные построения похожи на непрерывные построения за исключением того, что они налагают ограничения на количество построений, выполняемых в течение определенного периода времени. Например, последовательное построение может быть сконфигурировано на выполнение только один раз каждые пять минут, а не каждый раз, когда кто-то фиксирует изменение. Фиксируемые разработчиками изменения в рамках пятиминутного периода накапливаются до тех пор, пока этот период не истечет и не выполнится следующее построение.
- **Построения с условным возвратом.** Построения с условным возвратом также похожи на непрерывные построения, но вместо выдачи сигнала в ситуации, когда кто-то фиксирует критическое изменение, условные возвраты служат для запрета критических фиксаций еще до их достижения кодовой базы. Условные возвраты могут выполняться один раз на фиксацию (подобно непрерывным построениям) или же можно установить предел на количество их запусков в заданный промежуток времени (как в случае последовательных построений).
- **Запланированные построения.** Запланированные построения выполняются согласно специальному расписанию и не привязываются явно к действию фиксации. Наиболее популярный пример подхода запланированных построений известен как *ночные построения*, поскольку они запланированы на запуск в одно и то же время по ночам, после того как команда разработки сделала всю дневную работу. Так как эти типы построений не привязываются напрямую к действию фиксации, в целом для них приемлемо быть немножко устаревшими и не отражать наиболее актуальный код в кодовой базе. Запланированные построения также могут занимать больше времени при выполнении, возможно, запуская более глубокие автоматизированные тесты или создавая артефакты, такие как установочные пакеты, которые должны производиться только в ограниченных количествах.

Основной темой, связанной с разнообразными типами автоматизированных построений, является определение, насколько часто они запускаются, и сколько времени занимает каждый запуск построения до своего завершения. Работа, которую выполняет каждый тип построения, представляет собой расширение этого.

По мере того как построения становятся менее частыми, они требуют больше времени на завершение своих задач и, следовательно, могут выполнять большее количество задач с возрастающей сложностью и потребляемым временем.

Например, непрерывные построения сосредоточены на выполнении минимального объема работы для верификации текущего качества кодовой базы, в то время какочные и еженедельные построения могут требовать нескольких часов и даже дней на выполнение массовых задач, таких как всесторонний комплект глубоких автоматизированных тестов, компиляция крупных порций документации или пакетирование комплекта продуктов для выпуска.



Лучший подход к автоматизации построения обычно включает несколько разных типов построений, оперирующих в одно и то же время, каждое со своим приоритетом.

К примеру, для одного приложения могут быть реализованы три разных построения.

1. Непрерывное построение для проверки качества каждого возврата.
2. Последовательное построение, которое происходит не чаще одного раза в час и выполняет более детальные автоматизированные тесты, но требует более длительного времени на завершение.
3. Ночное построение, которое публикует изменения, внесенные за день, на тестовом веб-сайте, чтобы пользователи или команда контроля качества могли отследить продвижение работ и сообщить ошибках как можно раньше.

Создание автоматизированного построения

Выяснив тип построения, который желательно создать, можно приступить к его определению. Для выполнения своей работы сервер построения требует нескольких вещей: во-первых, сценария построения, содержащего задачи, которые сервер построения должен выполнить; и, во-вторых, исходный код (и другие артефакты), для которых будет выполняться сценарий построения.

Чтобы определить построение, использующее Team Foundation Server, выберите элемент Builds (Построения) на вкладке Team Explorer (Проводник команды), затем щелкните на ссылке New Build Definition (Новое определение построения). Это приведет к открытию мастера нового определения построения (New Build Definition Wizard), позволяющего сконфигурировать новое построение.

В этом примере мы создаем непрерывное построение, поэтому введите Continuous в поле Build definition name (Имя определения построения) на вкладке General (Общие), как показано на рис. 18.1, и оставьте остальные установки неизменными.

Далее выберите переключатель Continuous Integration (Непрерывная интеграция) на вкладке Trigger (Триггер), показанной на рис. 18.2. Вкладки Workspace (рабочее пространство) и Build Defaults (Стандартные параметры построения) можно пропустить, потому что предлагаемые на них стандартные установки вполне подходят. Вместо этого щелкните на вкладке Process (Процесс) и найдите свойство конфигурации Items to Build (Элементы для построения), как показано на рис. 18.3.

Свойство конфигурации Items to Build содержит список файлов проектов MSBuild, которые построение будет выполнять. Visual Studio автоматически выбирает текущее решение, предполагая, что необходимо, как минимум, все скомпилировать.

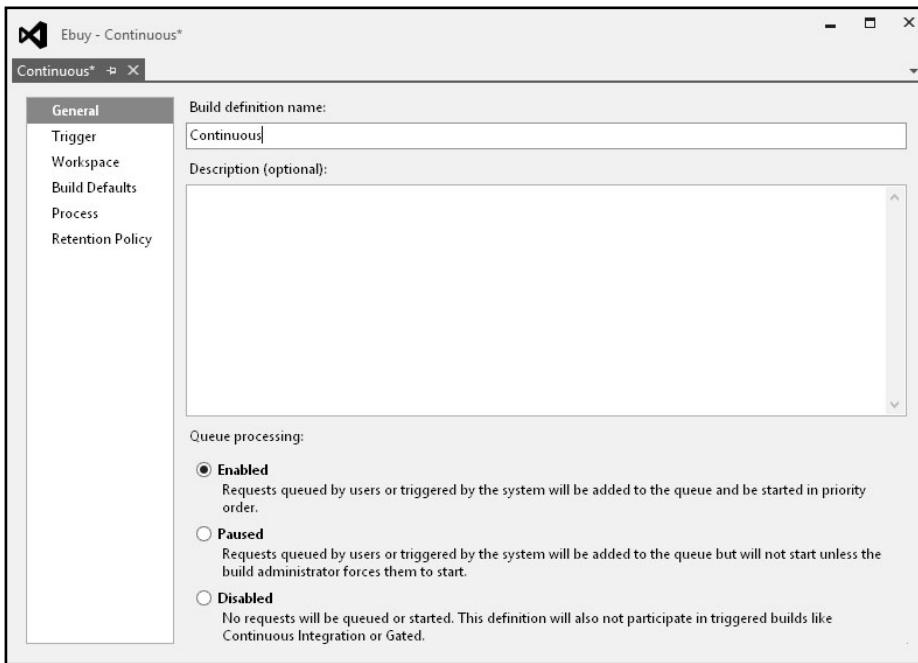


Рис. 18.1. Мастер нового определения построения – вкладка General

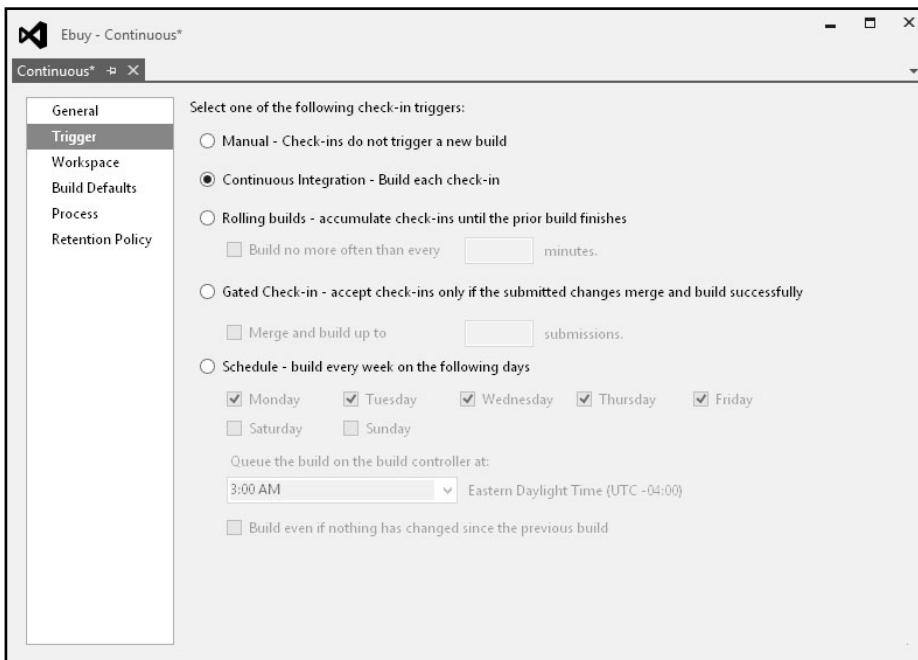


Рис. 18.2. Мастер нового определения построения – вкладка Trigger

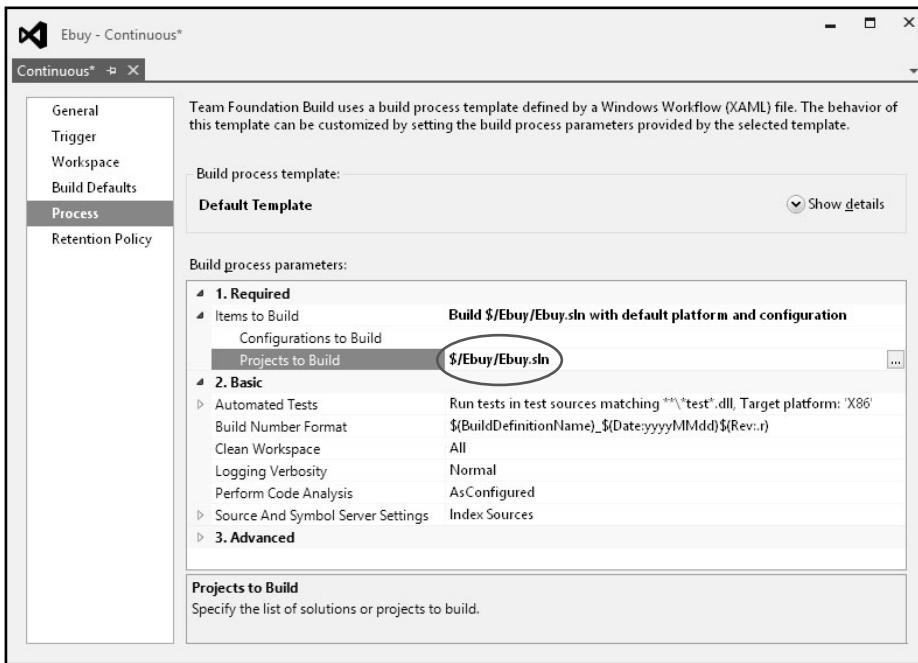


Рис. 18.3. Мастер нового определения построения – вкладка Process

Используйте это свойство конфигурации для выбора любых дополнительных файлов проектов MSBuild, созданных с целью предоставления дополнительной логики построения. В случае указания нескольких файлов проектов MSBuild будет выполнять их в порядке предоставления, останавливаясь при любом отказе, который может возникнуть.

Также обратите внимание на свойство конфигурации Automated Tests, стандартное поведение которого заключается в выполнении всех автоматизированных тестов, обнаруживаемых в любых сборках с именами, которые соответствуют выражению `***test*.dll`. Это стандартное значение означает, что в случае создания любых проектов автоматизированных тестов, в именах которых содержится слово `test`, сервер Team Foundation Server будет автоматически выполнять их без дополнительного конфигурирования.

Завершив конфигурирование нового определения построения, сохраните определение, как и любой другой файл (например, нажав `<Ctrl+S>`). После этого оно должно быть видно в разделе All Build Definitions (Все определения построений) на вкладке Team Explorer (Проводник команды), как показано на рис. 18.4.

Чтобы увидеть это в работе, попробуйте зафиксировать набор изменений в системе управления исходным кодом или щелкните правой кнопкой мыши на новом построении, выберите в контекстном меню пункт Queue New Build... (Поставить в очередь новое построение...) и щелкните на Queue (Очередь) в диалоговом окне. Сервер Team Foundation Server запустит построение, извлекая исходный код решения из системы управления исходным кодом и выполняя указанные сценарии построения.

При некотором везении построение должно завершиться успешно, указывая на то, что все было сделано правильно. Если же во время построения возникает сбой, исправьте любые ошибки, которые вы найдете в журналах.

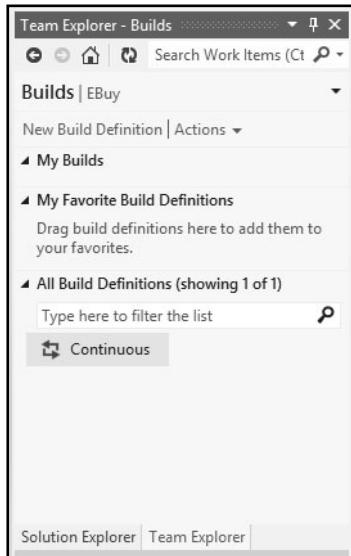


Рис. 18.4. Мастер нового определения построения – вкладка *Team Explorer*

При непрерывной интеграции всем членам команды рекомендуется проверять свои изменения, отправляемые в централизованный репозиторий системы управления исходным кодом, рано и часто, регулярно интегрируя свою работу с работой других членов команды. Это помогает избежать ситуации, когда отдельные разработчики модифицируют свои локальные копии приложения, но не фиксируют изменения в централизованном репозитории. Со временем различия между основной версией приложения и локальной копией каждого разработчика становятся настолько объемными, что требуют нескольких часов на согласование отличий. Если разработчики выполняют объединение рано и часто, время интеграции распределяется малыми дозами по всем возвратам, радикально сокращая свое воздействие.

Обнаружение проблем

Рассмотрим распространенный “жизненный цикл разработки программного обеспечения”: требования собраны, программное обеспечение построено, и готовое программное обеспечение протестировано для проверки корректности его построения. Эти шаги повторяются вплоть до выпуска программного продукта. Фундаментальная проблема этого подхода связана с тем, что зачастую проходит немало времени с момента, когда ошибка внесена на этапе разработки, до момента, когда она обнаружена на этапе тестирования — это приводит к возрастанию стоимости исправления ошибки.

Существует простая причина: когда вы только что завершили реализацию функциональной возможности, то еще хорошо помните написанный код, и это в целом позволит довольно легко найти и исправить проблему. С другой стороны, если между написанием кода и нахождением ошибки проходит значительный промежуток времени, можно не только подзабыть код, но также и пройти через несколько итераций, которые только усугубляют проблему. Таким образом, имеет смысл заняться проблемами как можно раньше.

При следующем возврате будет автоматически инициировано новое построение.

Поздравляем, вы создали свое первое работающее построение!

Непрерывная интеграция

Хотя непрерывная интеграция может выглядеть вполне рядовой в приведенном выше списке типов автоматизированных построений, в действительности она является чем-то большим, нежели просто типом автоматизированного построения, которое выполняется каждый раз, когда кто-то фиксирует изменение в коде.

На самом деле *непрерывная интеграция* (continuous integration – CI) также описывает процесс обеспечения качества приложения за счет реализации небольших, целенаправленных наборов функциональности и интеграции этой функциональности в более крупное приложение на регулярной основе. Такой подход не только сокращает время отклика, позволяя вводить новые возможности максимально быстро; он также служит намного более важной роли в обнаружении и исправлении проблем в кодовой базе сразу же после их возникновения.

Принципы непрерывной интеграции

Ниже приведен список из 10 ключевых принципов, которые имеют решающее значение для эффективного внедрения непрерывной интеграции. Для получения выигрыша от непрерывной интеграции может не требоваться реализация их всех, но получаемые преимущества обычно пропорциональны числу принципов, которые есть возможность соблюсти.



Эти принципы обычно приписывают Мартину Фаулеру, одному из организаторов движения гибкой (Agile) разработки программного обеспечения. За дополнительными сведениями о непрерывной интеграции (и многим другим вопросам) обращайтесь к обширному набору статей на его веб-сайте (<http://martinfowler.com/articles/continuousIntegration.html>).

Поддерживайте единственный репозиторий исходного кода

Системы управления исходным кодом предоставляют централизованный репозиторий, который позволяет членам команды эффективно разделять исходный код и другие артефакты. Системы централизованного управления исходным кодом также обеспечивают “источник правды” – централизованный репозиторий является одним местом, содержащим последнюю работающую версию приложения.

Автоматизируйте построение

Вы должны иметь возможность выполнять построение целого приложения на любой заданной машине за два шага.

1. Извлечь исходный код из репозитория системы управления исходным кодом.
2. Запустить единственную команду.

Чтобы все работало, потребуется обеспечить наличие в репозитории системы управления исходным кодом всего необходимого для построения, кроме основных системных зависимостей (таких как средства операционной системы, службы базы данных или платформа .NET Framework). Ничего другого для построения и выполнения приложения устанавливаться не должно. Когда дополнительные установки неизбежны, они должны быть включены как часть процесса автоматизации, которая инициируется во время выполнения единственной команды.

Это правило особенно касается баз данных. Всякий раз, когда это возможно, локальные построения не должны полагаться на существование удаленных баз данных; взамен автоматизированные построения должны включать схему базы данных и начальные данные для создания базы с нуля. Предпочтительно, чтобы это достигалось без необходимости в установке чего-либо на локальной машине (например, использовать встроенную версию Microsoft LocalDb, а не собственную установку Microsoft SQL Server или SQL Server Express).

Сделайте построение самопроверяемым

Это та точка, где многие шаблоны и практические приемы, изученные в этой книге, собираются вместе. Приемы SOLID и слабо связанная архитектура формируют путь для создания быстрых и эффективных модульных тестов, которые делают возможной очень частую проверку кода.

Процесс непрерывной интеграции может затем выполнять комплекты автоматизированных тестов применительно к любому возврату исходного кода с целью обна-

ружения проблем сразу же после их введения. Когда в ходе построения непрерывной интеграции какой-то модульный тест не проходит, узкий контекст отказавшего теста в сочетании с относительно небольшим количеством изменений исходного кода существенно облегчают нахождение и исправление проблемы.

Построение непрерывной интеграции, не умеющее верифицировать качество своего вывода, не имеет большой ценности.

Заставляйте каждого разработчика часто фиксировать изменения в главном репозитории

Для решения поставленных задач разработчики должны получить локальную копию исходного кода. С течением времени количество изменений, сделанных разработчиком в своей локальной копии, увеличивается, равно как и растет разрыв между локальной копией и главным репозиторием системы управления исходным кодом (“источником правды”). Если позволить такому продолжаться, отличия станут настолько огромными, что для восстановления интеграции этих двух исходных кодов потребуется прикладывать значительные усилия.

Таким образом, важно, чтобы все члены команды избегали ситуаций, при которых локальные репозитории становятся не синхронизированными с главным репозиторием, фиксируя изменения в главном репозитории настолько часто, насколько это возможно — не реже одного раза в день, а лучше чаще. В таком случае любой член команды может оставаться синхронизированным и главный репозиторий системы управления исходным кодом всегда предлагает точное представление текущего состояния проекта.

Каждая фиксация должна запускать построение главного кода на машине интеграции

Заявление “На моей машине это работает!” можно часто слышать в командах разработчиков по всему миру. Эта фраза отражает ситуацию, когда один разработчик все правильно сконфигурировал на своей индивидуальной рабочей станции и приложение работает отлично. С одной стороны, это хороший знак того, что средство разработано или ошибка исправлена; с другой стороны, упомянутая фраза является очень плохой приметой, указывая на то, что новое средство или исправление ошибки может вообще не заработать на любой другой машине.

Чтобы решить эту проблему, непрерывная интеграция требует настройки машины (или множества машин), называемой *машиной (машинами) интеграции*, которая компилирует и выполняет любую фиксацию в основной ветви кодовой базы. Машина интеграции должна максимально близко отражать производственную среду, поэтому можно предполагать, что если приложение компилируется и выполняется на машине интеграции, то есть отличный шанс, что оно будет делать то же самое и в производственной среде.

Имея машину интеграции, проверяющую каждую фиксацию, факт функционирования приложения на машине конкретного разработчика роли не играет — приложение должно запускаться на машине интеграции, и если это не так, то построение (и само приложение) должно считаться “не удавшимся” и немедленно быть исправлено.

Сохраняйте построение быстрым

Так как обнаружение проблем максимально близко к моменту их внесения является, пожалуй, наиболее решающим аспектом непрерывной интеграции, крайне важно, чтобы построения непрерывной интеграции выполнялись быстро, сообщая о любых

возникших проблемах. Время — деньги: чем больше времени проходит с момента введения проблемы до момента ее обнаружения, тем дороже обойдется ее исправление.

Тестируйте в копии производственной среды

Когда приложение тестируется в среде, не совпадающей со средой, в которой ожидается выполнение выпуска, выявление ошибок становится менее вероятным.

Обеспечьте для всех легкость получения актуального исполняемого модуля

Непрерывная интеграция способствует более тесной обратной связи, позволяя пользователям принимать активное участие в разработке приложений.

Наличие возможности доступа к действительным сборкам из последнего построения обычно очень полезно, но в случае веб-приложений иногда даже лучше развертывать их на централизованном тестовом сервере, чтобы любой мог тестировать приложения и оставлять отзывы.

Каждый может видеть, что происходит

Возможно, вам приходилось слышать вековой философский вопрос: слышен ли звук падающего дерева в лесу, если рядом никого нет? Эквивалентный вопрос в области разработки программного обеспечения выглядит так: если построение не удалось, и никто об этом не знает (или не переживает по этому поводу), имеет ли это хоть какое-то значение?

Состояние непрерывного построения имеет прямое отношение к общей работоспособности кодовой базы и, возможно, целого проекта. Это выходит далеко за рамки очевидной метрики — прошло построение или же потерпело неудачу, создавая такие метрики, как время, необходимое для завершения построения, и степень покрытия кода, происходящая во время построения. По этой причине очень важно, чтобы каждый участник проекта имел доступ ко всем деталям автоматизированных построений, и уделял пристальное внимание тому, как они выполняются.



Люди придумали много креативных путей для сохранения команд в курсе состояния построения. Эти идеи простираются от постоянного уведомления посредством приложений в системном лотке до больших телевизионных систем, смонтированных на стенах, чтобы любой мог их видеть.

Здесь важна сама осведомленность о построениях, а не способ получения этой информации. Поэтому выберите подходящий метод и уделяйте ему должное внимание.

Самая важная для понимания концепция, касающаяся непрерывной интеграции, заключается в следующем: когда построение нарушается, это значит, что *ваше приложение в каком-то отношении не работает*, и нет ничего важнее его исправления. При нарушении построения команда должна отложить все текущие дела и заняться исправлением проблемы с построением (т.е. исправлять приложение) — наиболее высокоприоритетной работой.



Не удавшееся построение должно всегда останавливать процесс. Если вы постоянно сталкиваетесь с не удавшимися построениями, которые не являются результатом нарушения работы приложения (например, они могут быть вызваны сбоями сетевых подключений, ошибками доступа к API-интерфейсам третьих сторон и т.д.), вы должны всерьез подумать о способе реорганизации построений или о рефакторинге тестов, чтобы избежать таких ложно позитивных результатов.

В подобных обстоятельствах непрерывное построение больше не является точным отражением работоспособности кодовой базы, и его ценность стремительно снижается. Хуже того, может начаться привыкание к не удавшимся построениям с потерей должного к ним внимания.

Не удавшееся построение должно всегда указывать на проблему с кодовой базой и никогда не становиться настолько частым явлением, чтобы возник облазн его игнорировать.

Автоматизируйте развертывание

Последний принцип непрерывной интеграции касается обеспечения простоты использования приложения целевой аудиторией. Подобно тому, как разумно проводить тестирование в клоне производственной среды, успешные построения непрерывной интеграции должны всегда включать автоматизированное развертывание приложения в среде, которая максимально близко имитирует производственную среду.

Автоматизированное развертывание дает два основных преимущества. Во-первых, оно позволяет все время видеть и использовать последнюю версию приложения, существенно облегчая ее тестирование на предмет ошибок и верификацию после их исправления.

Во-вторых, максимально частое выполнение автоматизированного развертывания приложения в среде, аналогичной производственной, распространяет ценность непрерывной интеграции далеко за рамки проверки того, что приложение работает, добавляя проверку того, что работает также и процедура *развертывания* приложения. Подобно тому, как автоматизированные тесты могут помочь в выявлении ошибок сразу после их внесения в кодовую базу, автоматизированное развертывание может вскрыть проблемы с процедурой развертывания (такие как отсутствующие зависимости или сложности с полномочиями безопасности), как только они появляются.

Хотя избежать проблем с развертыванием не удастся, по крайней мере, его можно испытать вне производственной среды, задолго до того, как наступит время финального выпуска. Это означает, что вы будете точно знать, что должно происходить при помещении приложения в производственную среду, поскольку вы уже столкнулись и решили проблемы с развертыванием, которые привели бы к задержке производственного выпуска. Вместо исключения теперь они являются просто еще одним шагом в (автоматизированном) процессе развертывания.



За дополнительными сведениями об автоматизации развертывания обращайтесь в главу 19.

Резюме

В настоящей главе было показано, как использовать шаблоны и практические приемы, изученные на протяжении этой книги, для сокращения объема человеческих усилий, вкладываемых в жизненный цикл разработки программного обеспечения, за счет автоматизации максимального количества задач из этого жизненного цикла. После вложений в такие вещи, как приемы разработки SOLID и комплект автоматизированных тестов, проверяющих приложение, обеспечивать непрерывное качество приложения будет чрезвычайно легко.

Выход в свет

В этой части...

Глава 19. Развертывание

Развертывание

Вы можете потратить массу времени, используя инфраструктуру ASP.NET MVC для построения самого лучшего веб-сайта в мире, однако он не сможет принести кому-либо пользу до тех пор, пока не будет размещен на веб-сервере, чтобы пользователи могли действительно к нему обращаться. Действие по копированию веб-сайта на веб-сервер и его открытие пользователям называется *развертыванием*, и оно определенно не является концепцией, уникальной для веб-сайтов ASP.NET MVC.

В этой главе мы раскроем несколько наиболее популярных приемов, позволяющих поместить веб-сайт в Интернет – от простого копирования файлов до работы с поставщиками “облачного” хостинга для обеспечения максимальной масштабируемости и бесперебойной работы.

При чтении этой главы имейте в виду, что многие веб-приложения имеют уникальные потребности в отношении развертывания, и рассматриваемые здесь приемы могут оказаться неприменимыми напрямую в конкретной ситуации. Вместо того чтобы рассматривать эту главу как руководство в стиле “как сделать то-то и то-то”, попробуйте думать о ней как об обзоре разнообразных доступных инструментов развертывания и приготовьтесь к поиску средств и технологий, применимых к вашей ситуации.

Что необходимо развертывать

Перед тем как начать создание веб-сайтов и копирование файлов, давайте сделаем шаг назад и обсудим, что именно мы собираемся делать. На высоком уровне существуют три вида зависимостей, которые имеются в большинстве веб-приложений: сборки .NET и различные файлы, которые содержат логику для сайта; любой специальный контент (такой как файлы CSS или JavaScript), на который опирается сайт; и любой тип внешних зависимостей времени выполнения, которые требуются веб-сайту (наподобие базы данных или внешних служб).

Основные файлы веб-сайта

Как минимум, каждое веб-приложение ASP.NET должно включать папку `/bin`, содержащую сборки с компилированным кодом приложения и другие сборки .NET, от которых зависит приложение. Таким образом, папка `/bin` является важной частью любой стратегии развертывания веб-приложений ASP.NET.

Однако также существуют и разнообразные “специальные” файлы, которые не обязательно требуются для корректного функционирования сайта, но часто содержат важную информацию наподобие конфигурации сайта. Эти файлы, такие как `web.config` и `Global.asax`, почти всегда должны быть включены в дополнение к сборкам в папке `/bin`.

Если вы уже являетесь разработчиком ASP.NET Web Forms, то ничего из сказанного не должно быть сюрпризом, поскольку к настоящему моменту было описано не что иное, как развертывание традиционного веб-приложения ASP.NET.

Тем не менее, развертывание веб-приложений ASP.NET MVC начинает отличаться от развертывания традиционных приложений ASP.NET Web Forms, когда дело доходит до представлений. В дополнение к сборкам приложения в папке /bin и любым “специальным” связанным с ASP.NET файлам, веб-приложения ASP.NET MVC должны также включать локальные копии всех своих представлений (т.е. папку /Views) вместе с остальным развертываемым контентом. Причина в том, что представления ASP.NET MVC поддерживают те же самые процедуры оперативной (Just-In-Time – JIT) компиляции, развертывания и сопровождения, что и представления .aspx из Web Forms; на самом деле, они практически синонимичны.

Библиотеки ASP.NET MVC для развертывания в папке /bin

Возможно, это само собой разумеется, но для работы веб-сайта ASP.NET MVC развернутому приложению будет необходим доступ к сборкам ASP.NET MVC Framework. Это можно решить двумя способами: установить ASP.NET MVC Framework прямо на веб-сервере или включить библиотеки ASP.NET MVC вместе с остальными сборками в папку /bin приложения.

Шаги по установке ASP.NET MVC Framework на сервере совпадают с показанными в первой главе этой книги – просто запустите Web Platform Installer (<http://www.microsoft.com/web/downloads/platform.aspx>) прямо на сервере и установите пакет ASP.NET MVC Framework.

Часто предпочтительнее избегать установки чего-либо на веб-сервере, поэтому сборки ASP.NET MVC можно трактовать подобно любым другим зависимостям приложения и копировать их в папку /bin вместе со всем остальным. Такой прием часто называют *развертыванием в папку /bin*, и этот подход обычно является наиболее простым, надежным и легким в сопровождении.

К счастью, Visual Studio существенно упрощает развертывание в папку /bin, включая пункт меню, который автоматически добавляет к приложению папку, содержащую зависимости ASP.NET MVC: просто щелкните правой кнопкой мыши на проекте ASP.NET MVC в Visual Studio и выберите в контекстном меню пункт Add Deployable Dependencies... (Добавить развертываемые зависимости...), как показано на рис. 19.1.

После этого Visual Studio позволит выбрать зависимости для включения в проект. Отметьте флагок ASP.NET MVC (рис. 19.2) и щелкните на кнопке OK.

Затем Visual Studio создаст новую папку по имени /_bin_deployableAssemblies, которая содержит сборки инфраструктуры, предназначенные для включения в развертывание (рис. 19.3).

Теперь при опубликовании веб-сайта Visual Studio будет развертывать сборки, находящиеся в этой новой папке, вместе с остальными частями приложения.

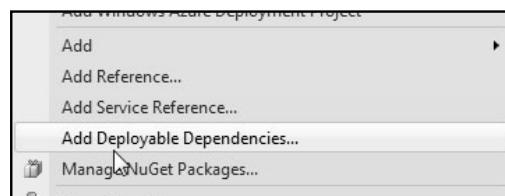


Рис. 19.1. Пункт меню для добавления в приложение папки, содержащей зависимости ASP.NET MVC

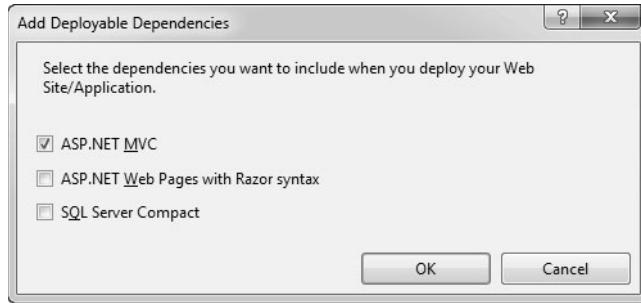


Рис. 19.2. Выбор ASP.NET MVC

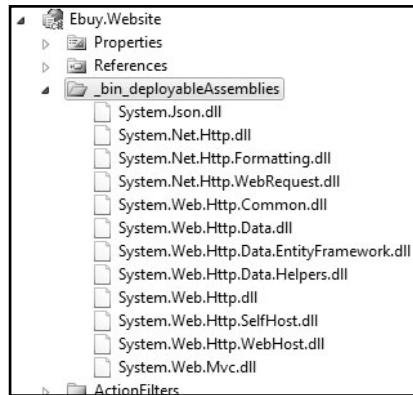


Рис. 19.3. Новая папка _bin_deployableAssemblies

Статический контент

Статическим контентом может считаться файл любого типа, но в большинстве случаев это будут в основном файлы JavaScript, стилей CSS и изображений, которые поддерживают клиентскую логику и стилизацию приложения. Хотя формально эти файлы могут находиться где угодно в структуре папок сайта, стандартные шаблоны проектов ASP.NET MVC создают папки /Scripts, /Images и /Content для помещения в них всех файлов JavaScript и другого контента. Следовательно, если вы используете такое соглашение, эти три папки будут содержать весь статический контент вашего сайта.



Не забудьте установить свойство Build Action (Действие построения) для каждого файла статического контента внутри проекта Visual Studio в Content (Контент), чтобы сообщить среде Visual Studio о том, что это статический контент, который должен быть развернут вместе с сайтом.



Рис. 19.4. Традиционная иерархия развертывания ASP.NET MVC

Что не должно развертываться

Если вы следете соглашениям, определенным в стандартных шаблонах проектов ASP.NET MVC, то почти каждое развертывание веб-сайта ASP.NET MVC будет выглядеть подобно иерархии каталогов, показанной на рис. 19.4.

Обратите внимание, что эта структура каталогов весьма отличается от набора файлов, с которыми вы работаете в проекте Visual Studio приложения (взгляните на пример проекта, приведенный на рис. 19.5).

Более конкретно, развернутое приложение не будет включать файлы исходного кода, которые определяют логику приложения. Поскольку перед развертыванием проект компилируется, эти файлы исходного кода уже “включены” в развертывание в форме скомпилированных сборок внутри каталога `/bin`. Имея это в виду, вы можете создать любую структуру папок для хранения и упорядочивания файлов исходного кода, т.к. эти папки не станут частью развернутого приложения.

Базы данных и другие внешние зависимости

Хотя почти каждое развертывание приложения ASP.NET MVC будет включать структуру каталогов, похожую на показанную на рис. 19.5, именно здесь заканчиваются все сходства между развертываниями приложений ASP.NET MVC и начинаются уникальные потребности в отношении процесса развертывания.

Большинство веб-приложений будут также зависеть от других вещей, отличных от физических файлов, которые разворачиваются на сервере, скажем, от возможности сохранения и извлечения данных из базы или взаимодействия с какой-то веб-службой. Такие зависимости только возрастают, по мере того как приложение становится менее тесно связанным и распределенным, а также с ростом популярности архитектур, ориентированных на службы. Несмотря на то что исследование деталей о том, как координировать развертывание веб-сайта с упомянутыми системами, выходит далеко за рамки настоящей книги, это не касается высокоуровневого обсуждения относительно планирования таких видов развертывания. Ниже приведен список вопросов, которые помогут обнаружить зависимости и задачи, требуемые для развертывания приложения.

1. Какие системные приложения и API-интерфейсы требуются приложению (например, версия операционной системы, версия IIS, версия .NET Framework)?
 - Должно ли любое программное обеспечение быть установленным на сервере?
2. Какие системные папки или файлы требуются приложению?
 - Требует ли приложение специфического пути папки для чего-либо? (Это то, чего обычно следует избегать.)
3. Требует ли приложение базы данных?
 - Если это так, имеются ли какие-то обновления схемы базы данных с момента последнего выпуска?
 - Имеет ли приложение дело с отдельным пользователем базы данных? Если да, то корректно ли сконфигурирован доступ в базу данных для этого пользователя?
4. С какими другими серверами или службами взаимодействует приложение?
 - Требуется ли вносить какие-то изменения в настройки сети для доступа к ним (например, правила брандмауэра, безопасность на основе пользователей или ролей)?
5. Все ли необходимые лицензии приобретены и доступны?

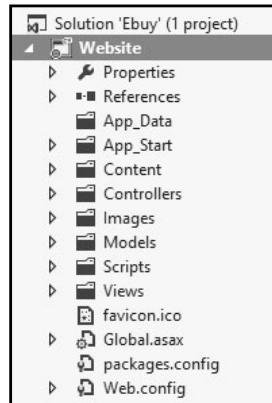


Рис. 19.5. Стандартная структура проекта ASP.NET MVC

Обратите внимание, что этот список определенно не включает в себя все, что должно быть принято во внимание для обеспечения успешного выпуска. Тем не менее, такие вопросы должны помочь в распространенных ситуациях и заставить задуматься о любых дополнительных требованиях, которые может иметь приложение.

Что требуется для приложения EBiu

Чтобы привести пример обдумывания процесса развертывания веб-сайта, давайте посмотрим, какие зависимости и конфигурации существуют при развертывании образцового приложения EBiu.

- **Системные API-интерфейсы и службы.** Приложение EBiu является достаточно элементарным приложением, которое не зависит ни от каких системных API-интерфейсов кроме .NET 4.5 Framework. Все другие зависимости от API-интерфейсов, включая ASP.NET MVC 4, предоставляются в папке /bin (которая, конечно же, должна быть развернута).
- **Представления, сценарии, таблицы стилей и изображения.** В дополнение к зависимостям от сборок, таким как .NET Framework и сборка, включающая логику приложения, указанные артефакты являются наиболее очевидными зависимостями, которые требуются для функционирования приложения. Мы должны обеспечить копирование этих файлов вместе со всеми другими.
- **База данных.** Веб-сайт EBiu управляет данными, поддерживаемыми моделью данных Entity Framework Code First, которая требует базы данных для хранения информации приложения.
- **Место для хранения загруженных изображений.** Когда пользователи создают новый список аукционных товаров, они имеют возможность загружать изображения элементов для отображения в списке, и приложение должно сохранять где-нибудь эти изображения. Действительное местоположение и метод сохранения файлов изображений может варьироваться в зависимости от того, где приложение размещено – на одиночном сервере, на ферме серверов или с помощью службы “облачного” хостинга наподобие Azure. Независимо от того, где хранятся файлы изображений, приложение должно иметь как “физический” доступ (т.е. доступ через сеть или файловую систему) к местоположению, так и соответствующие полномочия безопасности для чтения и записи файлов изображений.

После получения ответов на все указанные выше вопросы и выяснения всего того, что должно быть развернуто для корректного функционирования приложения, можно приступать к собственно развертыванию.

Развертывание на сервере IIS

Пожалуй, наиболее распространенный сценарий хостинга приложений ASP.NET MVC предусматривает создание и конфигурирование веб-сайта с использованием Internet Information Server (IIS). Хорошая новость заключается в том, что приложения ASP.NET MVC, по большей части, похожи на любые другие приложения ASP.NET, поэтому если вы уже умеете развертывать веб-приложения ASP.NET на сервере IIS, то изучать вам тут нечего, и ни один из приведенных ниже шагов не должен показаться незнакомым. Если же вы впервые имеете дело с веб-сайтами IIS или приложениями ASP.NET, то не переживайте – в последующих разделах будет представлено все, что необходимо для этого знать.

Предварительные условия

Прежде чем можно будет развертывать веб-сайт, необходимо убедиться, что целевой веб-сервер удовлетворяет всем предварительным условиям, которые необходимы для размещений приложения ASP.NET MVC. В ранних версиях .NET Framework соблюдение всех предварительных условий и развертывание приложения ASP.NET на веб-сервере требовало выполнения довольно большого количества шагов.

К счастью, в настоящее время все изменилось и единственным предварительным условием является установка на веб-сервере помимо самого IIS платформы .NET Framework (4.0 или последующей версии).

Развертывание сборок ASP.NET MVC Framework

Сами сборки ASP.NET MVC 4 также должны быть доступны, и для их развертывания имеется два варианта.

1. Запуск установщика ASP.NET MVC 4, как было описано в главе 1.
2. Включение сборок ASP.NET MVC Framework в папку /bin приложения, используя упомянутый ранее в этой главе метод развертывания в папку /bin.

Если на единственном сервере планируется запуск нескольких веб-сайтов ASP.NET MVC 4, имеет смысл выбрать первый вариант: установить ASP.NET MVC 4 один раз и больше не беспокоиться по этому поводу. Тем не менее, нет никаких веских причин следовать этому пути, кроме экономии дискового пространства, которое будут занимать сборки ASP.NET MVC Framework в каждом приложении.

Почти в каждом сценарии рекомендуется выбирать второй вариант и развертывать сборки ASP.NET MVC Framework в папке /bin приложения вместе с любыми другими сборками, от которых зависит приложение. Развертывание сборок вместе с приложением существенно упрощает его управление, сопровождение и даже обновление каждого веб-сайта изолированно, не беспокоясь о том, что изменение на уровне сервера может повлиять на другие сайты.

Создание и конфигурирование веб-сайта IIS

Создание нового веб-сайта IIS – очень прямолинейный процесс. Для начала создайте каталог для размещения веб-сайта, к примеру, C:\inetpub\wwwroot\Ebuy. Затем откройте диспетчер служб IIS, щелкните правой кнопкой мыши на элементе Default Web Site и выберите в контекстном меню пункт Add Application... (Добавить приложение...), как показано на рис. 19.6, чтобы отобразить диалоговое окно Add Application (Добавление приложения).

В этом диалоговом окне (рис. 19.7) введите имя веб-сайта (например, Ebuy) и путь к каталогу, созданному на первом шаге (т.е. C:\inetpub\wwwroot\Ebuy).

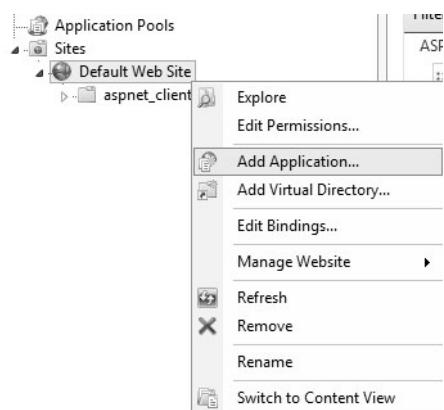


Рис. 19.6. Создание нового веб-сайта IIS

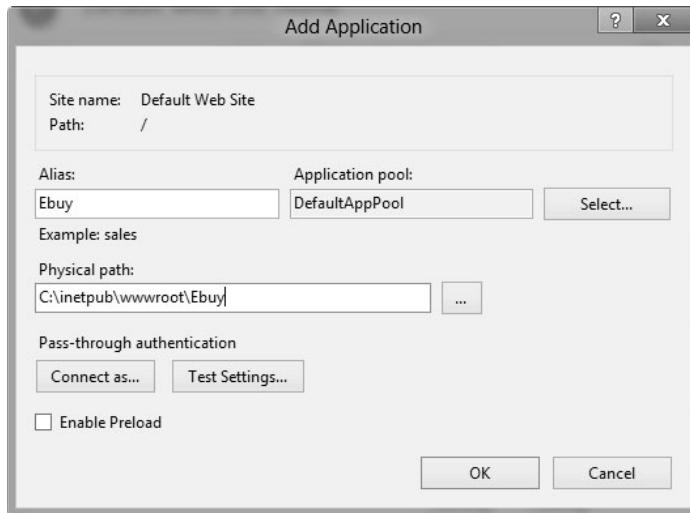


Рис. 19.7. Диалоговое окно Add Application

Остальные установки в этом диалоговом окне можно смело оставить без изменений, но ради эксперимента щелкните на кнопке Select... (Выбрать...) рядом с полем Application pool (Пул приложений) для открытия диалогового окна Select Application Pool (Выбор пула приложений) и удостоверьтесь, что стандартный пул приложений использует версию 4.0 платформы .NET Framework (как показано на рис. 19.8).

Если стандартный пул приложений не сконфигурирован на работу с версией 4.0 платформы .NET Framework, создайте новый пул приложений, который использует .NET 4.0. Если вы не видите версию 4.0 в списке доступных версий .NET Framework, это может означать, что платформа .NET Framework не была корректно установлена. Попробуйте переустановить .NET Framework и при необходимости запустите команду %FrameworkDir%\%FrameworkVersion%\aspnet_regiis.exe для правильного конфигурирования .NET Framework внутри IIS.

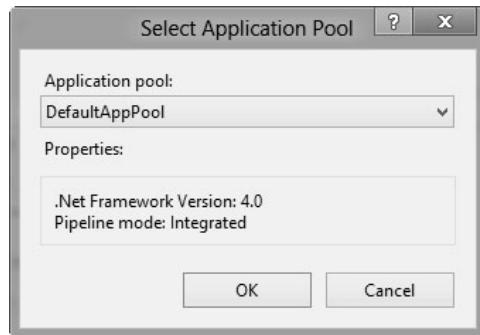


Рис. 19.8. Стандартный пул приложений сконфигурирован на использование .NET Framework версии 4.0

Наконец, щелкните на кнопке OK, чтобы заставить IIS создать ваш веб-сайт. Теперь можно приступать к развертыванию своего сайта.



Предыдущие версии ASP.NET MVC, размещенные в IIS 6, требовали выполнения специальных шагов конфигурирования, чтобы разрешить маршрутизацию URL без расширений посредством ASP.NET MVC. Однако это больше не является проблемой, потому что ASP.NET 4 конфигурирует IIS для маршрутизации чего-либо, не имеющего расширения, прямо в ASP.NET. Обратите внимание, что если вы имеете дело с сервером IIS 7 или IIS 7.5, функционирующим под управлением Windows Vista SP2, Windows Server 2008, Windows Server 2008 R2 SP2 или Windows 7, потребуется применить к системе обновление (<http://support.microsoft.com/kb/980368>).

Публикация из Visual Studio

После того, как приложение создано и сконфигурировано в IIS, на выбор доступно несколько технологий развертывания.

Самой доступной технологией развертывания является встроенный в Visual Studio механизм публикации. Для его использования щелкните правой кнопкой мыши на проекте ASP.NET MVC и выберите в контекстном меню пункт Publish... (Публиковать...), как показано на рис. 19.9, чтобы открыть мастер Publish Web (Публикация в веб).

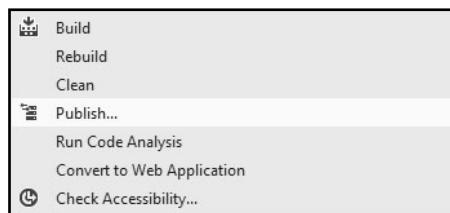


Рис. 19.9. Открытие мастера Publish Web в Visual Studio

Для создания нового профиля публикации, который позволит развернуть ваш веб-сайт, выберите элемент <New...> (<Новый...>) в раскрывающемся списке на вкладке Profile (Профиль) и назначьте новому профилю имя (например, Local IIS Website). Так как развертывание выполняется в локальную файловую систему, далее выберите в списке Publish method (Метод публикации) вариант File System (Файловая система), как показано на рис. 19.10.

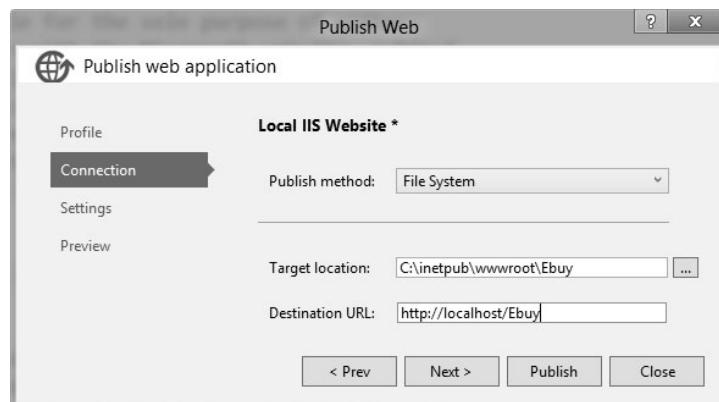


Рис. 19.10. Мастер Publish Web



Вариант публикации **File System** подходит только при развертывании на веб-сервере, к файловой системе которого вы имеете прямой доступ через сеть. Если же применяется служба веб-хостинга, это может не походить, потому придется выбрать подход публикации посредством FTP для развертывания веб-сайта через универсальный протокол FTP, поддерживаемый всеми основными веб-хостами.

После выбора метода публикации диалоговое окно изменится, чтобы позволить ввести другую конфигурационную информацию, которая необходима Visual Studio для публикации с применением выбранного метода. Можно даже сохранить несколько конфигураций публикации, выбирая имя профиля в списке профилей публикации и щелкнув на кнопке **Save** (**Сохранить**).

Сконфигурировав все требуемые варианты методов публикации, щелкните на кнопке **Publish** (**Публиковать**) и Visual Studio развернет ваш сайт в указанном местоположении. После успешного развертывания веб-сайта Visual Studio автоматически откроет браузер и перейдет на вновь развернутый сайт.

Если вы пытались проделать эти шаги на своей машине, развертывание, скорее всего, потерпит неудачу – это было сделано намеренно, чтобы показать, каким образом диагностировать проблемы с развертыванием! При развертывании сайта Visual Studio фиксирует все, что делается, в окне **Output** (**Вывод**). Если во время развертывания возникают какие-то проблемы, они должны быть здесь отображены.

Например, развертывание, которое вы пытались только что выполнить, может завершиться неудачей из-за того, что вы не имеете доступа в папку `C:\inetpub\wwwroot\Ebuy`. Если это так, в окне **Output** должно появиться сообщение **ACCESS DENIED** (доступ запрещен). Чтобы исправить такую ошибку, измените параметры безопасности на целевой папке, включив доступ по записи для текущего пользователя, и попробуйте опубликовать сайт снова. На этот раз публикация должна пройти успешно и в окне браузера отобразится развернутый сайт.

Копирование файлов с помощью MSBuild

Как было показано в главе 18, неплохо автоматизировать максимально возможную часть процесса разработки приложений – и вряд ли где-либо это будет настолько справедливым, как при развертывании.

Хотя механизм публикации Visual Studio довольно удобен, необходимость открытия среды Visual Studio каждый раз, когда требуется развертывать сайт, становится достаточно утомительной. Таким образом, давайте посмотрим, как можно воспроизвести все то, что делает мастер **Publish Web** в Visual Studio, с помощью автоматизированного сценария MSBuild.

В следующем примере представлен сценарий MSBuild, который сначала собирает решение, используя задачу построения MSBuild, а затем копирует файлы веб-приложения в целевой каталог веб-сайта с помощью задачи **Copy**:

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0" DefaultTargets="Deploy"
      xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <PropertyGroup>
        <BuildDir>$ (MSBuildProjectDirectory) \build\</BuildDir>
    </PropertyGroup>

    <Target Name="Deploy">
        <MSBuild Projects="EBuy.sln" Properties="OutDir=$ (BuildDir)" />
    </Target>
</Project>
```

```

<ItemGroup>
    <WebsiteFiles Include="$(BuildDir)\_PublishedWebsites\Ebuy.Website\**" />
</ItemGroup>
<Copy SourceFiles="@{WebsiteFiles}"
      DestinationFiles=
"@{WebsiteFiles->$(DeploymentDir)\%{RecursiveDir}\%{Filename}\%{Extension}'}"
      SkipUnchangedFiles="true"
    />
</Target>
</Project>

```

Чтобы выполнить этот сценарий, откройте окно командной строки Visual Studio, перейдите в папку решения Ebu и введите следующую команду:

```
msbuild.exe deploy.proj /p:DeploymentDir=" [Путь к целевой папке]"
```

Эта команда запустит построение приложения и направит вывод во временный каталог построения (build в текущем каталоге), после чего скопирует содержимое папки _PublishedWebsites, созданной MSBuild для веб-приложений, в целевую папку по вашему выбору.

Выполнение сценариев базы данных с помощью MSBuild

Механизмы публикации в файловой системе Visual Studio и развертывания MSBuild отлично работают для развертывания файлов, но как поступить в ситуации, когда приложение зависит от базы данных, которая не может быть развернута с помощью простого копирования файлов?

Одним из великолепных средств Entity Framework Code First является возможность автоматического управления версиями базы данных – вы можете сообщить инфраструктуре о необходимости обновления базы данных автоматически во время фазы запуска сайта, всякий раз, когда она видит, что произошло изменение модели, требующее изменения схемы базы данных. Однако если вы не пользуетесь средством Entity Framework Code First, то ответственность за отслеживание и развертывание любых изменений в схеме базы данных, которые могут возникать на протяжении разработки, возлагается на вас.

Когда речь заходит о развертывании изменений базы данных, обычно доступны два варианта.

1. Повторное создание целой базы данных каждый раз.
2. Сохранение каждого обновления базы данных в отдельном файле сценария и выполнение этих файлов для приведения целевой базы данных к актуальному состоянию согласно последней схеме.

Естественно, вариант 1 является самым простым решением во время разработки – всегда проще построить базу данных с нуля, чем беспокоиться об обновлении существующей базы данных. Тем не менее, если только вы не разрабатываете первый производственный выпуск, такой подход не соответствует финальному производственному развертыванию и, следовательно, не отвечает духу непрерывной интеграции: как можно более частое тестирование производственного развертывания с целью обнаружения проблем на самом раннем этапе.

Если убрать вариант 1, то вариант 2 – множество файлов сценариев с инкрементными изменениями схемы базы данных – де-факто становится единственным выбором. К счастью, развертывание с помощью любого из этих подходов осуществляется довольно просто с применением MSBuild и утилиты *SQLCMD* из SQL Server.

Чтобы добавить к построению выполнение SQL-сценариев, дополните файл MSBuild следующими строками MSBuild:

```
<Target Name="DeployDatabase">
  <ItemGroup>
    <ScriptFiles Include="$(ScriptsDir)\*.sql" />
  </ItemGroup>
  <Exec Command="sqlcmd -E -S $(SqlServer) -i \"$(ScriptFilesFullPath)\" />
</Target>
```

Эти несколько строк ищут SQL-сценарии в указанном пути (`$(ScriptsDir)*.sql`) и для каждого найденного файла .sql запускают утилиту SQLCMD в отношении экземпляра SQL Server, сконфигурированного в свойстве `$(SqlServer)`. Обратите внимание, что эти сценарии выполняются в порядке их обнаружения утилитой MSBuild. Это будет порядок, в котором они встречаются в файловой системе – по именам файлов – поэтому часто полезно применять какое-то подходящее соглашение об именовании, такое как предварение каждого имени файла со сценарием числовым префиксом.

После этого можно запустить следующую команду (она должна располагаться в одной строке), чтобы заставить MSBuild выполнить SQL-сценарии и построить базу данных автоматически:

```
msbuild.exe deploy.proj /t:DeployDatabase
/p:ScriptsDir=Scripts /p:SqlServer=.\SQLEXPRESS
```



Утилита SQLCMD устанавливается как часть стандартной установки Microsoft SQL Server, но для ее использования не обязательно иметь установленную копию Microsoft SQL Server. В качестве альтернативы можно установить бесплатный продукт Microsoft SQL Server Feature Pack. Загрузить и установить последнюю версию Microsoft SQL Server Feature Pack можно, воспользовавшись предпочтаемой поисковой системой для нахождения продукта по названию, или посредством следующей ссылки для Microsoft SQL Server 2008 R2 SP1 Feature Pack: <http://www.microsoft.com/en-us/download/details.aspx?id=26728>.

Развертывание в Windows Azure

Если вы хотите избежать традиционного хостинга своего веб-сайта и воспользоваться преимуществами облачных технологий, доступен еще один вариант развертывания и хостинга с применением платформы облачного хостинга от Microsoft, которая называется Windows Azure. С помощью Windows Azure можно сосредоточиться на своем приложении и позволить Microsoft заботиться об инфраструктуре, требуемой для его размещения в Интернете.

В оставшейся части этого раздела мы рассмотрим простые шаги для развертывания приложения в облаке с использованием Windows Azure. По завершении вы получите публичный веб-сайт, размещенный в облаке.

Создание учетной записи Windows Azure

Перед тем, как можно будет развернуть веб-сайт в облаке, используя Windows Azure, потребуется зарегистрировать учетную запись Windows Azure. Для этого посетите веб-сайт Windows Azure (<http://www.windowsazure.com>), найдите и щелкните на ссылке Free trial (Попробовать бесплатно), чтобы создать новую учетную запись.

После создания учетной записи вы получите доступ к порталу управления Windows Azure (Windows Azure Management Portal) — онлайновому порталу для управления службами облачного хостинга.

Создание нового веб-сайта Windows Azure

Чтобы создать новый веб-сайт с применением портала Windows Azure, щелкните на меню New (Новый) в нижней левой части страницы и выберите пункт Web Site (Веб-сайт), а затем щелкните на Create with Database (Создать с базой данных) для открытия мастера New Web Site (Новый веб-сайт), показанного на рис. 19.11.

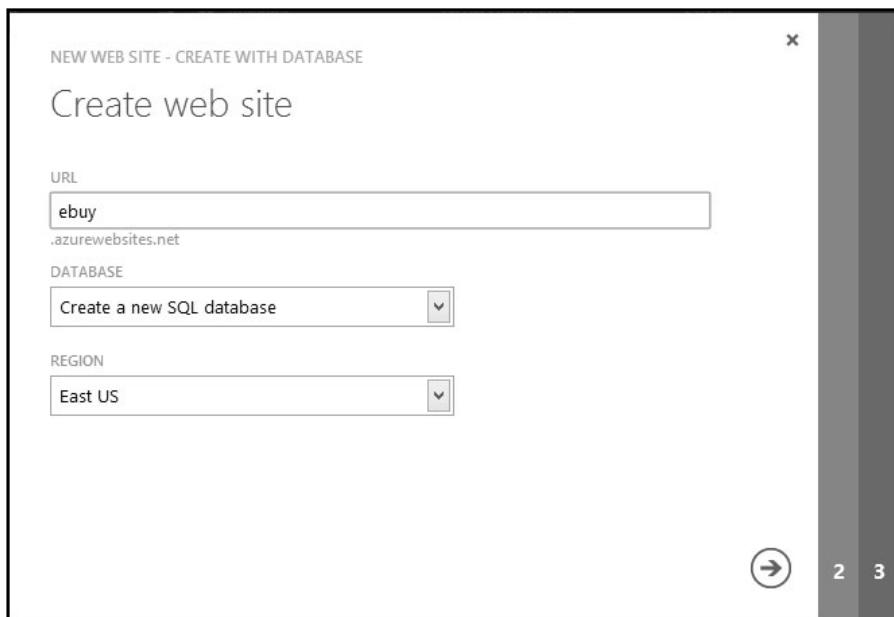


Рис. 19.11. Мастер New Web Site в Windows Azure

Введите сведения о новом веб-сайте, такие как DNS-имя сайта и регион (т.е. информационный центр), где сайт должен быть размещен. Поскольку приложение EBuy требует базы данных для хранения своей информации, выберите в раскрывающемся списке Database (База данных) вариант Create new SQL database (Создать новую базу данных SQL).

Следующие несколько шагов мастера помогут сконфигурировать новую базу данных; стандартные значения обычно подходят для большинства маленьких веб-сайтов. Завершив предоставление сведений о новом сайте и его базе данных, щелкните на ссылке Create Web Site (Создать веб-сайт) для создания нового сайта.

После предоставления Windows Azure определенного времени на создание и обеспечение работы нового веб-приложения щелкните на нем в списке приложений, чтобы приступить к управлению им.

Публикация веб-сайта Windows Azure через систему управления исходным кодом

Общепризнанным простейшим способом развертывания приложения на веб-сайте Windows Azure является использование встроенной поддержки для публикации через систему управления исходным кодом Team Foundation Server (TFS) или Git.

Поскольку в главе 18 было дано введение в сервер TFS, мы продолжим им пользоваться в рассматриваемом примере. Однако имейте в виду, что общий процесс будет тем же самым, который применяется с методом публикации через систему управления исходным кодом Git.

Чтобы приступить к использованию публикации через систему управления исходным кодом TFS, щелкните на ссылке Set up TFS publishing (Настроить публикацию TFS) в панели инструментов вашего веб-сайта Azure (рис. 19.12) для запуска мастера конфигурации системы управления исходным кодом TFS.

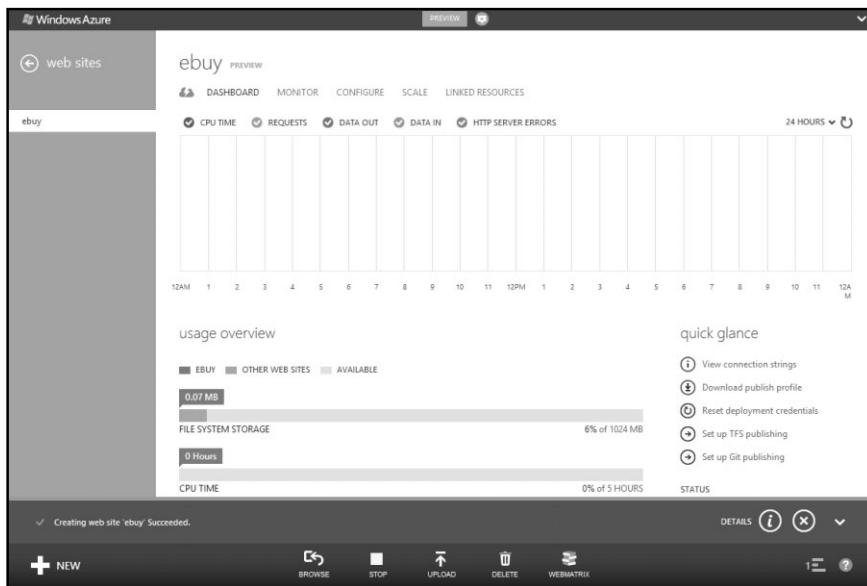


Рис. 19.12. Панель инструментов веб-сайта Azure

Затем введите имя пользователя, которое указывалось при создании учетной записи TFS Preview в предыдущей главе (или щелкните на ссылке для создания новой учетной записи TFS Preview, если это еще не делалось), и щелкните на ссылке Authorize now (Авторизовать сейчас), чтобы авторизовать Windows Azure для доступа к учетной записи TFS Preview.

После успешной авторизации Windows Azure перейдите в следующее диалоговое окно и выберите проект из системы управления исходным кодом, который необходимо соединить с этим веб-сайтом, затем щелкните на галочке для завершения. Спустя несколько секунд Windows Azure соединит веб-сайт с указанным проектом TFS.

При таком соединении каждый возврат, выполняемый в проекте TFS, будет запускать новое построение проекта в TFS. После каждого успешного построения заново собранный веб-сайт развертывается в Windows Azure, и все изменения становятся актуальными безо всяких дополнительных усилий с вашей стороны.

Непрерывное развертывание

Одно из основных преимуществ автоматизированного развертывания состоит в том, что развертывание приложения в любой среде становится невероятно простым и часто довольно быстрым. Следующий логический шаг после подготовки автоматизированного развертывания заключается в максимально частом его запуске — возможно, с каждым возвратом. Такой процесс очень частого развертывания приложения называется *непрерывным развертыванием*.

Непрерывное развертывание — отличный способ увеличения прозрачности проекта за счет предоставления любому заинтересованному лицу возможности узнать *точное* состояние приложения в текущий момент. Это позволяет пользователям опробовать новые функциональные средства в момент их возврата и начать предоставлять отклики о них на ранних этапах разработки. Считайте это ручной, управляемой человеком вариацией концепции непрерывной интеграции, которая ориентирована на выявление проблем на как можно более ранних этапах.

Если вы используете развертывание TFS с Windows Azure, как было показано в предыдущем разделе, то уже выполняете непрерывное развертывание. Но даже если вы не применяете Windows Azure с TFS, это вовсе не означает невозможность получить преимущества от непрерывного развертывания — вам просто придется проделать чуть больше работы для его настройки.

Чтобы добавить в проект механизм непрерывного развертывания, уделите некоторое время на выяснение, какие концепции из этой главы и главы 18 применимы к вашему проекту, и скомбинируйте их вместе для автоматизации всех шагов, требуемых для развертывания приложения.

При должной реализации непрерывное развертывание может быть отличным способом обеспечить развитие приложения за счет максимально быстрой передачи изменений в руки пользователей и получения ценных откликов на протяжении всего процесса разработки.

Резюме

Финальная задача при разработке любого вида программного обеспечения заключается в его передаче конечным пользователям. В случае веб-сайта ASP.NET MVC это может включать различные задачи, наиболее важной из которых является копирование файлов, запускающих приложение, в корректно сконфигурированный веб-сайт IIS, находящийся в локальной сети или размещенный где-то в облаке.

Существует много способов развертывания приложений ASP.NET MVC — от встроенного в Visual Studio механизма публикации до специального автоматизированного развертывания, использующего MSBuild — и ни один из них не подходит для абсолютно любого приложения, поэтому важно найти подход, который будет работать для вашего проекта.

После того, как вы построили свой идеальный механизм публикации, попытайтесь его автоматизировать, чтобы значительно упростить как свою работу, так и работу тех, кому придется развертывать ваше приложение в будущем.

ЧАСТЬ VI

Приложения

В этой части...

Приложение А. Интеграция ASP.NET MVC и Web Forms

Приложение Б. Использование NuGet в качестве платформы

Приложение В. Рекомендуемые приемы

Приложение Г. Перекрестные ссылки: целевые темы,
функциональные возможности и сценарии

Интеграция ASP.NET MVC и Web Forms

Инфраструктура ASP.NET MVC Framework была не первым вторжением Microsoft в экосистему веб-разработки. В действительности это далеко не так. Предшественница ASP.NET MVC, инфраструктура ASP.NET Web Forms (которую после появления ASP.NET MVC называют просто ASP.NET), была представлена с первой версией платформы .NET Framework в начале 2002 года. В течение следующего десятилетия использование ASP.NET Web Forms Framework неуклонно росло, достигнув критической массы путем поддержки множества веб-сайтов в Интернете. Кроме того, значительное число разработчиков обрели хорошие навыки создания и обслуживания веб-сайтов ASP.NET Web Forms. Затем, спустя несколько лет, была выпущена инфраструктура ASP.NET MVC.

Существующие веб-сайты и обретенные навыки не могут просто исчезнуть или немедленно отбрасываться в сторону лишь потому, что появилась новая технология. На самом деле наоборот — многие из этих сайтов являются важными инвестициями, приносящими реальную и постоянную пользу бизнесу. В этом приложении рассматриваются различные концепции и стратегии, которые могут помочь ввести ASP.NET MVC Framework в существующие приложения ASP.NET Web Forms. Также будет описано несколько ловушек, которых следует избегать, чтобы сделать переход более гладким.

В следующих разделах будут продемонстрированы разнообразные приемы повышения уровня интеграции и сосуществования приложений ASP.NET MVC и Web Forms. Опробуйте их и выберите подходящие технологии для себя и своей команды.

Выбор между ASP.NET MVC и ASP.NET Web Forms

Несмотря на разделение общей платформы, во многих отношениях ASP.NET MVC и ASP.NET Web Forms представляют собой две конкурирующие инфраструктуры. Обе инфраструктуры помогают разработчикам ASP.NET быстро и эффективно доставлять основанные на веб решения, но делают они это своими уникальными способами.

Приложения Web Forms, как правило, не способствуют применению приемов проектирования SOLID, которые были описаны в главе 2. Это означает, что многие разработчики, предпочитающие подход SOLID, неохотно используют ASP.NET Web Forms для построения веб-приложений на основе .NET, но жаждут получить инфраструктуру, которая лучше подходит для их нужд. Когда эти разработчики столкнулись

с ASP.NET MVC, они тотчас же заметили, что данная инфраструктура удовлетворяет их потребностям, позволяя более эффективно реализовать принципы SOLID, и они немедленно отдали ей предпочтение.

Если вы вместе с командой не входите в упомянутую группу разработчиков — т.е. концепции и технологии, описанные в этой книге, не побудили вас к использованию этой новой инфраструктуры — то, возможно, ASP.NET MVC не является правильным вариантом для вашего проекта. В таком случае совершенно нормально продолжать работу с Web Forms, не переходя на ASP.NET MVC. Инфраструктура Web Forms определенно никуда не денется; на самом деле с каждым выпуском .NET Framework она продолжает обретать дополнительную и улучшенную функциональность.



Прежде чем принять решение переходить на новую инфраструктуру, удостоверьтесь, что вы и команда понимаете и согласны с фундаментальными концепциями, которые управляют ASP.NET MVC (они были описаны в главе 2), такими как архитектура SOLID.

Переход приложений или команд из Web Forms в ASP.NET MVC без такого глубокого понимания, просто ради использования “последней наилучшей инфраструктуры”, может повлечь за собой катастрофические последствия. Поскольку эти две инфраструктуры настолько тесно связаны и разделяют одну и ту же платформу, очень легко писать “приложение Web Forms”, применяя принципы Web Forms в инфраструктуре ASP.NET MVC Framework. Именно поэтому зачастую очень важно, чтобы разработчики не имели предыдущего опыта работы с Web Forms: предшествующие навыки не мешают использованию концепций MVC.

Имейте это в виду, если вы или члены вашей команды являются опытными разработчиками Web Forms, и постоянно проверяйте написанный код, чтобы убедиться в следовании шаблону MVC, а не возврату к способам, принятым Web Forms.

Перевод сайта Web Forms на ASP.NET MVC

Для совершенно новых приложений обычно рекомендуется избрать одну инфраструктуру и неуклонно ее придерживаться. Тем не менее, при наличии существующего приложения Web Forms, которое необходимо перенести в ASP.NET MVC, решение не обязательно является взаимоисключающим. Причина того, что в предыдущем разделе эти две инфраструктуры описаны как конкурирующие друг с другом, также является и причиной, которая делает возможным их интеграцию в единственном приложении: они обе построены поверх платформы ASP.NET.

Вспомните, каким образом запросы обрабатываются в приложении Web Forms: сервер IIS получает запрос, выясняет, как он отображается на физический файл (страницу .aspx) в структуре папок сайта, и затем запускает HTTP-обработчик Web Forms для выполнения страницы. А теперь вспомните об обработке запросов в ASP.NET MVC. Сервер IIS получает запрос, но не ищет соответствующий физический файл, а обращается к таблице маршрутов, в которой указано, что запрос должен быть обработан HTTP-обработчиком MVC, после чего запускает этот обработчик. А теперь подумайте о том, что таблица маршрутов является основным средством ASP.NET, которое работает в приложениях Web Forms так же хорошо, как и приложениях ASP.NET MVC!

Концепции сосуществования ASP.NET MVC и Web Forms в одном и том же приложении очень легко понять, если взглянуть на то, что на самом деле вы создаете не

“приложение ASP.NET MVC” (на основе контроллеров и представлений) или “приложение Web Forms” (на основе страниц .aspx), а *приложение ASP.NET* на основе HTTP-модулей и HTTP-обработчиков. Если рассматривать приложение в этих терминах, то обеспечение совместной работы двух инфраструктур сводится просто к размещению файлов и конфигурации!

В следующих разделах описано несколько приемов, которые делают переход от Web Forms на ASP.NET MVC намного менее трудоемким, позволяя извлечь максимум из существующих вложений в Web Forms. Тем не менее, все эти приемы основаны на одной фундаментальной концепции: сервер IIS должен иметь возможность выяснить, чем является текущий запрос – запросом ASP.NET MVC или запросом Web Forms. Как только это определено, IIS может отправить запрос подходящему обработчику, и приложение будет вести себя ожидаемым образом.

Добавление ASP.NET MVC к существующему приложению Web Forms

Если вы совершенно не склонны к риску и хотели бы внести в приложение минимально возможные изменения, можете дополнить существующее приложение Web Forms функциональностью ASP.NET MVC. Другими словами, вы не даете инфраструктуре ASP.NET MVC возможность “вступить во владение” вашим приложением, а позволяете только обрабатывать определенные очень специфичные запросы. Используя этот подход, вы определяете, какие запросы будут обрабатываться ASP.NET MVC, за счет регистрации специальных правил маршрутизации.

Чтобы заставить существующее приложение маршрутизировать запросы в ASP.NET MVC, сначала понадобится выполнить ряд шагов конфигурирования.

1. Добавьте сборки `System.Web.Mvc` и `System.Web.Razor` в список ссылок на сборки в приложении.
2. Добавьте в коллекцию `assemblies` элемента `compilation` внутри элемента `system.web` файла `web.config` следующие записи:

```
System.Web.Mvc, Version=4.0.0.0, Culture=neutral,  
PublicKeyToken=31BF3856AD364E35
```

```
System.Web.WebPages, Version=2.0.0.0, Culture=neutral,  
PublicKeyToken=31BF3856AD364E35
```

3. Создайте папку `/Views` в корне приложения. Местоположение этой папки внутри приложения важно, потому что фабрика представлений будет просматривать те же самые пути к физическим файлам, что и обычное приложение ASP.NET MVC.

- Как это делалось бы в стандартном приложении ASP.NET MVC, создайте папку `/Views/Shared` для хранения общих представлений.
- Скопируйте файл `/Views/web.config` из существующего веб-сайта ASP.NET MVC или воспользуйтесь файлом, содержимое которого показано в примере А.1. Этот конфигурационный файл очень важен, т.к. он регистрирует обработчик типа файлов Razor и сообщает IIS о том, что файлы под этой папкой предназначены только для внутреннего использования приложением и не должны быть доступны внешнему миру (в случае их запроса IIS должен возвращать ошибку 404 Not Found (не найдено)).

4. Дополнительно создайте папку Controllers для хранения контроллеров ASP.NET MVC. В отличие от папки /Views, местоположение контроллеров ASP.NET MVC не имеет значения, поэтому вы можете поместить эту папку куда угодно. Формально вы даже могли бы разместить контроллеры в совершенно другом проекте.

5. Наконец, потребуется добавить конфигурацию маршрутизации в класс, который определяет HttpApplication (обычно файл Global.asax.cs), как это делается в стандартном приложении ASP.NET MVC. С технической точки зрения вы применяете тот же самый API-интерфейс для регистрации маршрутов, которые будут обрабатываться порцией ASP.NET MVC сайта. Фундаментальная разница этого сценария заключается в том, что имеются две инфраструктуры, соперничающие за одни и те же URL, поэтому данный момент должен быть учтен при создании маршрутов.

В примере A.1 показан упомянутый ранее файл /Views/web.config.

Пример A.1. Конфигурационный файл /Views/web.config

```
<?xml version="1.0"?>
<configuration>
  <configSections>
    <sectionGroup name="system.web.webPages.razor"
      type="System.Web.WebPages.Razor.Configuration.RazorWebSectionGroup,
      System.Web.WebPages.Razor, Version=2.0.0.0, Culture=neutral,
      PublicKeyToken=31BF3856AD364E35">
      <section name="host"
        type="System.Web.WebPages.Razor.Configuration.HostSection,
        System.Web.WebPages.Razor, Version=2.0.0.0, Culture=neutral,
        PublicKeyToken=31BF3856AD364E35"
        requirePermission="false" />
      <section name="pages"
        type="System.Web.WebPages.Razor.Configuration.RazorPagesSection,
        System.Web.WebPages.Razor, Version=2.0.0.0, Culture=neutral,
        PublicKeyToken=31BF3856AD364E35"
        requirePermission="false" />
    </sectionGroup>
  </configSections>
  <system.web.webPages.razor>
    <host
      factoryType="System.Web.Mvc.MvcWebRazorHostFactory, System.Web.Mvc,
      Version=4.0.0.0, Culture=neutral,
      PublicKeyToken=31BF3856AD364E35" />
    <pages pageBaseType="System.Web.Mvc.WebViewPage">
      <namespaces>
        <add namespace="System.Web.Mvc" />
        <add namespace="System.Web.Mvc.Ajax" />
        <add namespace="System.Web.Mvc.Html" />
        <add namespace="System.Web.Routing" />
      </namespaces>
    </pages>
  </system.web.webPages.razor>
  <appSettings>
    <add key="webpages:Enabled" value="false" />
  </appSettings>
```

```
<system.web>
  <httpHandlers>
    <add path="*" verb="*" type="System.Web.HttpNotFoundHandler"/>
  </httpHandlers>

  <pages
    validateRequest="false"
    pageParserFilterType="System.Web.Mvc.ViewTypeParserFilter,
      System.Web.Mvc, Version=4.0.0.0, Culture=neutral,
      PublicKeyToken=31BF3856AD364E35"
    pageBaseType="System.Web.Mvc.ViewPage, System.Web.Mvc, Version=4.0.0.0,
      Culture=neutral, PublicKeyToken=31BF3856AD364E35"
    userControlBaseType="System.Web.Mvc.ViewUserControl, System.Web.Mvc,
      Version=4.0.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35">
    <controls>
      <add assembly="System.Web.Mvc, Version=4.0.0.0, Culture=neutral,
        PublicKeyToken=31BF3856AD364E35" namespace="System.Web.Mvc"
        tagPrefix="mvc" />
    </controls>
  </pages>
</system.web>

<system.webServer>
  <validation validateIntegratedModeConfiguration="false" />
  <handlers>
    <remove name="BlockViewHandler"/>
    <add name="BlockViewHandler" path="*" verb="*"
      preCondition="integratedMode" type="System.Web.HttpNotFoundHandler" />
  </handlers>
</system.webServer>
</configuration>
```

Имейте в виду, что физические файлы (например, страницы .aspx из Web Forms) будут всегда иметь преимущество перед любыми определяемыми маршрутами, а это означает, что порции Web Forms приложения будут выбираться в ущерб логике контроллеров ASP.NET MVC. Оставив это исключение в стороне, остальные рассмотренные в книге приемы можно применять для построения гибридного приложения Web Forms/MVC почти как “чистого” приложения ASP.NET MVC.

Копирование функциональности Web Forms в приложение ASP.NET MVC

Как было показано в предыдущем разделе, большая часть работы по обеспечению совместного функционирования ASP.NET MVC и Web Forms в одном приложении связана с конфигурированием аспектов, касающихся стороны ASP.NET MVC. Таким образом, имеет смысл – в зависимости от текущего приложения Web Forms – создать в качестве отправной точки новое приложение ASP.NET MVC и перенести в него существующие страницы Web Forms.

Хотя этого новое приложение будет сконфигурировано с учетом ASP.NET MVC, остается актуальным тот факт, что физические страницы .aspx будут иметь преимущество перед любой логикой маршрутизации ASP.NET MVC. Таким образом, оба подхода позволяют достигнуть одного и того же результата: функциональность Web Forms и ASP.NET MVC сосуществует на одном сайте.

Интеграция функциональности Web Forms и ASP.NET MVC

В нескольких предыдущих разделах описывались приемы, позволяющие обеспечить сосуществование ASP.NET MVC и Web Forms Frameworks внутри одного и того же приложения. Однако поскольку эти две инфраструктуры построены поверх платформы ASP.NET, обе они разделяют основную функциональность, которая дает им возможность выйти за рамки простого “существования”, чтобы действительно совместно использовать данные и функциональность – другими словами, *интегрироваться* друг с другом.

Управление пользователями

Пожалуй, наиболее важной – или, во всяком случае, чаще всего используемой – частью разделяемой функциональности является управление пользователями на основе поставщиков ASP.NET для аутентификации с помощью форм, аутентификации Windows, ролей, членства и профилей. Мало того, что эти поставщики продолжают отлично работать с приложением ASP.NET MVC, имеется даже возможность сохранения любого кода Web Forms (такого как страницы входа, страницы пользовательского профиля или администрирования и т.д.), который пользуется указанными поставщиками.

Например, когда страница Web Forms использует поставщик аутентификации с помощью форм для проверки и аутентификации пользователя, для этого пользователя генерируется маркер сеанса ASP.NET, который применяется основным API-интерфейсом ASP.NET для аутентификации пользователя в каждом запросе. Это означает, что пользователь может аутентифицировать себя через страницу Web Forms, а затем быть перенаправлен контроллеру ASP.NET MVC и по-прежнему считаться аутентифицированным.

Управление кешем

Еще одной широко используемой точкой интеграции является функциональность кеширования ASP.NET. Разработчики Web Forms пользовались классом System.Web.Caching.Cache уровня приложения (обычно доступного через свойство HttpContext.Cache) и классом System.Web.HttpSessionState уровня пользователя (доступном через свойство HttpContext.Session) для управления кэшированными данными еще со времен версии 1.1 платформы .NET Framework, и нет никаких веских причин прекращать это делать!

Подобно ранее упомянутым поставщикам управления пользователями, эти два класса являются частью основного API-интерфейса ASP.NET и одинаково доступны как разработчикам Web Forms, так и разработчикам ASP.NET MVC. Когда Web Forms и ASP.NET MVC сосуществуют на одном веб-сайте, они также разделяют процессы приложения, а это значит, что данные, записанные посредством API-интерфейсов Cache и HttpSessionState одной инфраструктурой, будут доступны по время последующих запросов к другой инфраструктуре.

Многое, многое другое!

Существует множество других API-интерфейсов, которые нормально взаимодействуют с ASP.NET MVC и Web Forms. Чтобы найти их, просмотрите документацию по API-интерфейсу и существующий код – почти любое пространство имен, которое не

начинается с `System.Web.UI`, является неплохим кандидатом на интеграцию между инфраструктурами.



Хотя многие части ASP.NET Framework доступны посредством как Web Forms, так и ASP.NET MVC, одной значительной порцией Web Forms Framework, *не* поддерживаемой в ASP.NET MVC, является состояние представления (`ViewState`). Состояние представления чаще всего используется при взаимодействии страницы Web Forms с самой собой, так что шансы столкнуться с проблемами, касающимися состояния представления, во время перекрестной отправки между страницами Web Forms и контроллерами ASP.NET MVC совсем невелики.

Тем не менее, при переводе приложения Web Forms в ASP.NET MVC будьте внимательны с любым кодом, который полагается на состояние представления — данные `ViewState` не будут существовать на протяжении запроса ASP.NET MVC, поэтому код, зависящий от этого, скорее всего, работать не будет!

Резюме

В этом приложении объяснялось, как “новая” инфраструктура ASP.NET MVC Framework вписывается в “унаследованные” приложения ASP.NET Web Forms.

Оказалось, что разделяемый ими API-интерфейс ASP.NET предоставляет впечатляющий путь обновления для переноса существующего кода Web Forms в ASP.NET MVC. На тот случай, когда существующий код не может быть легко обновлен или заменен, лежащая в основе платформа ASP.NET также поддерживает среду с преимущественно гладким сосуществованием, где эти две инфраструктуры могут не только функционировать бок о бок, но даже интегрироваться друг с другом.

Использование NuGet в качестве платформы

В главе 1 был представлен инструмент управления пакетами NuGet, который помогает устанавливать, конфигурировать и сопровождать разнообразные зависимости приложения, а в других главах этой книги приводились примеры потребления пакетов, опубликованных и поддерживаемых Microsoft и сообществом в целом. Тем не менее, вы не обязаны ограничивать себя пакетами, которые опубликовали другие разработчики.

В этом приложении будет предложено краткое введение в создание собственных пакетов и обзор того, чем в точности является пакет NuGet. Помимо этих основ вы найдете ряд советов и трюков по NuGet, которые могут помочь сделать разработку более приятной как для вас, так и для команды.



Цель этого приложения заключается не в том, чтобы научить вас всему, что известно о NuGet – собственная документация NuGet достаточно хороша, чтобы справиться с такой задачей! Вместо этого в приложении кратко представлены фундаментальные основы применения NuGet в качестве *инструмента*, а затем показано, каким образом можно использовать NuGet как *платформу*.

Установка инструмента командной строки NuGet

Хотя установочный пакет ASP.NET MVC устанавливает диспетчер пакетов NuGet (NuGet Package Manager) для работы с пакетами NuGet в разрабатываемых проектах, чтобы создавать и распространять собственные пакеты, сначала понадобится загрузить *инструмент командной строки NuGet* из сайта NuGet CodePlex (<http://nuget.codeplex.com/releases>).

Найдите загрузку, которая называется NuGet Command Line Bootstrapper (Начальный загрузчик командной строки NuGet) в разделе Downloads (Загрузки) сайта NuGet CodePlex, после чего загрузите и запустите его. Эта загрузка в действительности предоставляет доступ к начальному загрузчику – при первом его запуске будет произведено извлечение последней версии инструмента командной строки NuGet и замена его самого обновленной версией.

После загрузки и запуска начального загрузчика для обновления до последней версии инструмента командной строки NuGet переместите исполняемый модуль в папку, которая доступна из командной строки Visual Studio (к примеру, в каталог .NET Framework).

Итак, наступило время для создания пакетов!

Создание пакетов NuGet

Простейший способ создания пакета NuGet предусматривает запуск команды `nuget pack` в отношении существующего проекта Visual Studio. Например:

```
nuget pack MyApplication.csproj
```

Эта команда создаст пакет NuGet, используя версию сборки, имя проекта и другие метаданные, которые извлекаются из файла `AssemblyInfo.cs` проекта.

Файлы NuSpec

Файл NuSpec – это конфигурационный XML-файл, в котором указываются содержимое и метаданные пакета (например, идентификатор пакета, версия, имя, зависимости и т.д.). Этот файл требуется NuGet для любого создаваемого пакета, поскольку метаданные содержат важную информацию, которую NuGet использует для определения, какие пакеты – и какие их версии – необходимо загрузить для удовлетворения зависимости.

Это верно и при генерации пакета NuGet из файла проекта Visual Studio, как было показано в предыдущем примере. Даже если вы никогда этого не видели, NuGet на самом деле создает временный файл NuSpec, применяемый для генерации финального пакета NuGet.

Проблема с разрешением NuGet автоматически генерировать этот важный файл состоит в том, что вы теряете большую часть контроля над сгенерированным пакетом. Более того, возможно, вы намного лучше знаете, от чего должны зависеть ваши сборки, нежели то, что NuGet может определить на основе просмотра файла проекта.

Таким образом, предпочтительнее создать и настроить собственный файл NuSpec, а не позволить NuGet генерировать его самостоятельно. В последующих разделах показаны различные методы, которые можно использовать для создания и настройки файлов NuSpec.

Использование инструмента командной строки NuGet

Первый метод создания файла NuSpec является вариацией ранее применяемой команды: `nuget spec`. Команда `nuget spec` очень похожа на команду `nuget pack`, за исключением того, что `nuget spec` сохраняет сгенерированный файл NuSpec на диске, поэтому его можно продолжить модифицировать до использования при генерации финального пакета.

Например, в командной строке Visual Studio можно было бы запустить следующую команду, чтобы сгенерировать файл NuSpec для упомянутого выше файла проекта `MyApplication.csproj`:

```
nuget spec MyApplication.csproj
```

В качестве альтернативы ту же самую команду можно запустить в отношении предварительно построенной сборки:

```
nuget spec -a MyApplication.dll
```

Приведенные выше команды создают файл по имени `MyApplication.nuspec`, который выглядит следующим образом:

```
<?xml version="1.0"?>
<package>
  <metadata>
    <id>$id$</id>
    <version>$version$</version>
    <title>$title$</title>
    <authors>$author$</authors>
    <owners>$author$</owners>
    <licenseUrl>http://LICENSE_URL_HERE_OR_DELETE_THIS_LINE</licenseUrl>
    <projectUrl>http://PROJECT_URL_HERE_OR_DELETE_THIS_LINE</projectUrl>
    <iconUrl>http://ICON_URL_HERE_OR_DELETE_THIS_LINE</iconUrl>
    <requireLicenseAcceptance>false</requireLicenseAcceptance>
    <description>$description$</description>
    <releaseNotes>Summary of changes made in this release of the package.</releaseNotes>
    <copyright>Copyright 2012</copyright>
    <tags>Tag1 Tag2</tags>
  </metadata>
</package>
```

В начальном состоянии поля в сгенерированном файле NuSpec заполнены маркерами, соответствующими шаблону `$[имя]$`, который NuGet заменяет действительными значениями во время выполнения команды `nuget pack`.

Очевидно, этот шаблон не содержит какой-то специальной информации. Он просто определяет отправную точку, которую можно настраивать для отражения деталей конкретного проекта. В этот момент понадобится открыть файл NuSpec в предпочтитаемом редакторе XML (таком как редактор XML, встроенный в Visual Studio) и модифицировать файл вручную для определения, как должен быть сконфигурирован ваш пакет.

Использование проводника пакетов NuGet

В качестве альтернативы ручному редактированию XML-файлов NuSpec можно зайти на страницу загрузки NuGet (<http://nuget.codeplex.com/releases>) и загрузить *проводник пакетов NuGet* (NuGet Package Explorer). Помимо другой функциональности управления пакетами, проводник пакетов NuGet предоставляет великолепный графический пользовательский интерфейс, который помогает строить файлы NuSpec. Проводник пакетов NuGet несколько упрощает создание файла NuSpec.

1. Первым делом выберите вариант *Create New Package* (Создать новый пакет) на домашнем экране приложения (или нажмите `<Ctrl+N>`).
2. Затем выберите пункт меню *Edit⇒Edit Package Metadata...* (Правка⇒Редактировать метаданные пакета...) для создания нового проекта, который можно начать редактировать.
3. В этот момент проводник пакетов находится в режиме редактирования (рис. Б.1), и вы можете применять его графический пользовательский интерфейс для указания разнообразных аспектов ваших пакетов.
4. Завершив настройку пакета, можно выбрать пункт меню *File⇒Save* (Файл⇒Сохранить) или нажать комбинацию `<Ctrl+S>` для генерации и сохранения пакета NuGet на диске и/или выбрать пункт меню *File⇒Save Metadata As...* (Файл⇒Сохранить метаданные как...) для сохранения на диске файла NuSpec.

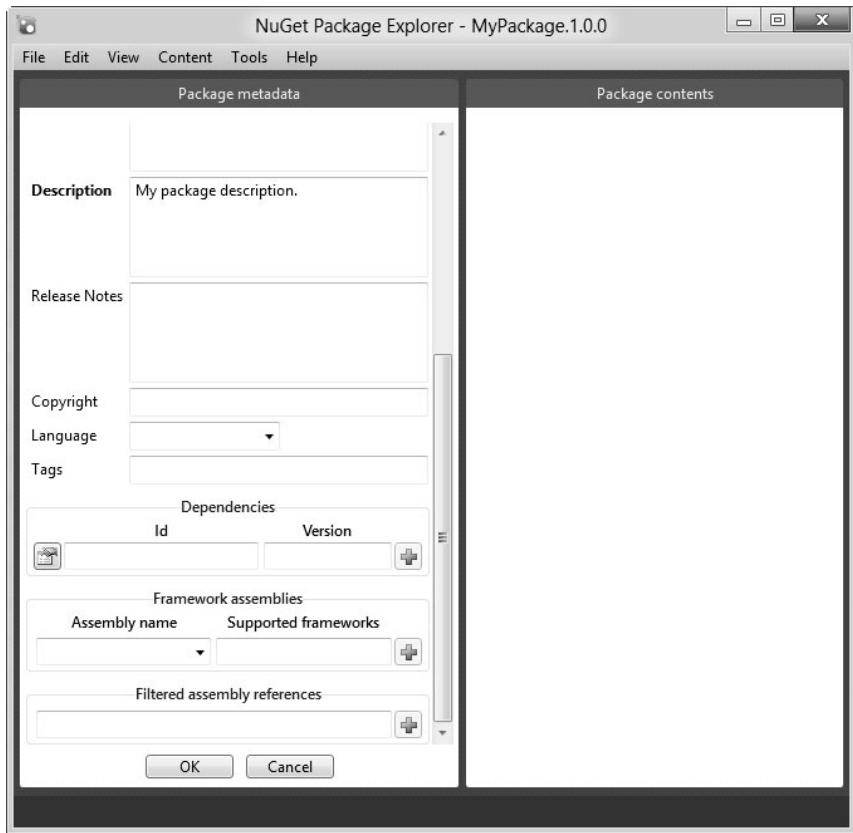


Рис. Б.1. Редактирование пакета с помощью проводника пакетов NuGet

Генерация пакета NuGet из файла NuSpec

Имея файл NuSpec, который определяет содержимое вашего пакета, можно воспользоваться командой `nuget pack` для генерации пакета NuGet.

Например, для генерации пакета NuGet из файла `MyApplication.nuspec`, созданного в предыдущем примере, служит следующая команда:

```
nuget pack MyApplication.nuspec
```

Эта команда сгенерирует новый пакет NuGet по имени `MyApplication.1.0.0.nupkg`, который содержит весь контент и сборки, указанные в файле NuSpec.

После этого данный пакет NuGet можно развернуть в репозитории пакетов NuGet для использования в своих приложениях.

Указание значений маркеров

Когда файл NuSpec содержит маркеры, такие как присутствуют в стандартном сгенерированном шаблоне, команда `nuget pack`, скорее всего, сообщит о том, что ей не известно, как их обрабатывать. Если это так, можно указать значения для этих маркеров, применяя переключатель `-Properties` и предоставив разделенный точками с запятой список пар “ключ/значение”.

Например, следующая команда (которая должна вводиться в одной строке) подставит вместо любых ссылок на маркер `$description$` фразу `My custom package description`:

```
nuget pack MyApplication.nuspec  
-Properties description="My custom package description"
```

Установка версии

Кроме того, команда `nuget pack` предоставляет переключатель `-Version`, который позволяет указать версию генерируемого пакета. Переключатель `-Version` может быть применен к любому файлу NuSpec независимо от того, указано ли в нем значение маркера для поля `Version` (Версия).

К примеру, приведенная ниже команда сгенерирует версию 1.7.0 пакета `MyApplication` независимо от указания значения свойства `version` в файле NuSpec:

```
nuget pack MyApplication.nuspec -Version 1.7.0
```

Структура пакета NuGet

Теперь, когда было показано, каким образом создавать собственные пакеты NuGet, давайте вернемся на шаг назад и проанализируем, что собой представляет сам пакет NuGet.

В сущности, пакеты NuGet – это просто воображаемые файлы ZIP, содержащие специальные метаданные (в форме файлов `.nuspec`) и некоторые или все следующие компоненты: сборки (также известные как “библиотеки”), контент и инструменты.

Если открыть пакет NuGet в предпочтаемой программе архивации, можноувидеть структуру папок, напоминающую показанную в примере Б.1.

Пример Б.1. Пример структуры папок NuGet

```
\Content  
    \App_Start  
        ConfigureMyApplication.cs.pp  
    web.config.transform  
    [Другие файлы контента и папки]  
\libs  
    \net40  
        MyApplication.dll  
    \sl4  
        MyApplication.dll  
    [Папки для других поддерживаемых инфраструктур]  
\tools  
    init.ps1  
    install.ps1  
    uninstall.ps1  
MyApplication.nuspec
```

Папка Content

Папка `Content` представляет собой корневую папку целевого приложения. Все, что помещено в эту папку – изображения, текстовые файлы, шаблоны классов или даже подпапки – будет скопировано в целевое приложение.

В дополнение к файлам, которые normally копируются, папка Content может также включать шаблоны трансформации конфигурационного файла и исходного кода. Они упрощают избирательную модификацию определенных частей целевого проекта.

В примере Б.1 можно было видеть трансформацию конфигурационного файла web.config.transform. Этот файл может включать следующий код:

```
<configuration>
  <system.webServer>
    <handlers>
      <add name="MyHandler" path="MyHandler.axd" verb="GET, POST"
           type="MyApplication.MyHandler, MyApplication"
           preCondition="integratedMode" />
    </handlers>
  </system.webServer>
</configuration>
```

Когда инструмент NuGet добавляет в проект пакет, содержащий этот файл web.config.transform, он обновит файл web.config данного проекта и добавит конфигурацию HTTP-обработчика MyHandler.

Пример Б.1 также содержит шаблон трансформации исходного кода App_Start\ConfigureMyApplication.cs.pp, который может иметь следующий вид:

```
[assembly: WebActivator.PreApplicationStartMethod(
  typeof($rootnamespace$.MyHandlerInitializer),
  "Initialize")]

namespace $rootnamespace$ {
  public class MyHandlerInitializer {
    public static void Initialize()
    {
      // Конфигурировать параметры MyHandler во время выполнения.
    }
  }
}
```

Как и при трансформации файла web.config, когда инструмент NuGet устанавливает этот пакет, он копирует ConfigureMyApplication.cs.pp в папку App_Start проекта, запускает трансформацию, после чего удаляет расширение .pp, создавая полностью функциональный класс, который может немедленно использоваться в проекте.

Папка libs

После папки Content следует папка libs. Назначение этой папки довольно прямо-линейно: любые содержащиеся в ней сборки добавляются в коллекцию References целевого проекта.

Сборки могут быть помещены в корень этой папки или, что лучше, в папку, специфичную для платформы, такую как net40, для указания, на какую платформу и версию эти сборки ориентированы. Разделяя сборки по различным папкам, можно эффективно нацеливаться на множество платформ и множество версий этих платформ посредством единственного пакета.

На момент написания этих строк инструмент NuGet распознавал три разных платформы, перечисленные в табл. Б.1.

Таблица Б.1. Платформы, распознаваемые NuGet

Платформа	Аббревиатура
.NET Framework	net
Silverlight	sl
.NET Micro Framework	netmf

В примере Б.1 эта функциональность демонстрировалась в действии за счет включения двух версий сборки `MyApplication.dll`: одна была ориентирована на .NET Framework версии 4.0 (`net40`), а вторая – на Silverlight версии 4 (`sl4`).

Папка tools

Наконец, рассмотрим папку `tools`. Эта папка содержит любые сценарии, исполняемые модули или другой контент, который разработчики могут быть заинтересованы использовать, но не включать в свои проекты в виде контента или ссылочных сборок. Папка `tools` может также содержать один или больше перечисленных ниже “специальных” сценариев PowerShell, которые NuGet ищет и выполняет при обработке каждого пакета.

- `init.ps1`. Запускается при первой установке пакета в решении.
- `install.ps1`. Запускается при каждой установке пакета.
- `uninstall.ps1`. Запускается при каждом удалении пакета.

Инструмент NuGet выполняет эти сценарии внутри контекста Visual Studio, предоставляя им полный доступ к API-интерфейсу DTE среды Visual Studio. Это позволяет применять такие сценарии для запрашивания и манипулирования практически чем угодно внутри Visual Studio каждый раз, когда пакет инициализируется, устанавливается или удаляется.

В дополнение к выполнению отмеченных выше специальных сценариев, NuGet будет добавлять любые папки `tools` к пути, доступному в консоли управления пакетами (Package Management Console), всякий раз, когда пакет устанавливается в решении. Это существенно облегчает распространение сценариев и исполняемых модулей, которые поддерживают активную разработку, но не должны развертываться с финальным приложением.

Например, пакет `MvcScaffolding` содержит несколько сценариев PowerShell, которые помогают разработчикам генерировать модели, представления и контроллеры в своих приложениях ASP.NET MVC. Эти сценарии PowerShell исключительно полезны для экономии времени и увеличения продуктивности, однако они предназначены для применения в процессе разработки, а не для поставки вместе с готовым продуктом.

Типы пакетов NuGet

Теперь, когда вы знаете, что именно пакет NuGet может содержать, давайте посмотрим, как можно использовать пакеты NuGet для своих целей.

На высоком уровне пакеты NuGet могут быть разделены на несколько категорий: *пакеты сборок*, *пакеты инструментов* и *метапакеты*. Хотя все они создаются с применением той же самой спецификации и управляются через NuGet, пакеты из каждой категории используются по очень разным причинам.

Пакеты сборок

Пакеты сборок – это пакеты, назначение которых заключается в добавлении к проекту одной или большего числа сборок, а также любого вспомогательного контента, который такие сборки требуют или ожидают. Пакеты сборок являются наиболее распространенными, т.к. они представляют собой основную причину создания инструмента NuGet.

Пакеты инструментов

Пакеты инструментов вводят инструменты в среду разработки для их использования при построении приложений. В этом контексте “инструменты” предназначены для оказания помощи при разработке и тестировании и обычно не являются частью финального выпуска приложений. Инструментами могут быть как простые сценарии PowerShell, так и полноценные приложения.

Метапакеты

Метапакеты – это пакеты, которые ссылаются на другие пакеты. Главное назначение метапакетов заключается в том, чтобы помочь в быстрой подготовке и запуске проекта за счет автоматической загрузки и конфигурирования нескольких зависимостей посредством установки одного пакета.

Например, теоретически мы могли бы создать пакет “EF Code First + ELMAH + Glimpse + Ninject”, который включает все пакеты, необходимые для написания об разового приложения из этой книги. После этого при выполнении всех примеров, рассмотренных в книге, вам понадобилось бы только выбрать пункт меню File⇒New Application... (Файл⇒Новое приложение...), указать в качестве типа проекта ASP.NET MVC 4 Web Application (Веб-приложение ASP.NET MVC 4) и воспользоваться NuGet для установки этого метапакета, получив сразу все необходимые ссылки.

Разделение пакетов NuGet

После создания пакета NuGet понадобится добавить его в *репозиторий пакетов*, чтобы распространить среди других разработчиков, позволив им потреблять его в своих приложениях. Когда дело доходит до распространения построенных вами пакетов, в сущности, на выбор доступны два варианта: публикация пакета в открытом репозитории пакетов NuGet.org или размещение собственного репозитория пакетов.

Публикация в открытом репозитории пакетов NuGet.org

Во время установки установщик NuGet заранее конфигурирует единственный репозиторий – открытый репозиторий пакетов NuGet, размещенный на веб-сайте NuGet.org. Поскольку он поступает предварительно сконфигурированным, открытый репозиторий пакетов NuGet.org является наиболее удобным способом разделения пакетов с другими разработчиками, и если созданный вами пакет содержит функциональность, которой вы хотите поделиться с миром, то загрузка этого пакета в открытый репозиторий NuGet.org представляется хорошей идеей.

Перед тем, как появится возможность опубликовать пакет в открытом репозитории NuGet.org, понадобится создать учетную запись на веб-сайте NuGet.org, посетив <http://nuget.org> и щелкнув на ссылке Register (Регистрация) вверху страницы.

Использование мастера загрузки пакетов NuGet.org

После создания учетной записи можно приступать к использованию открытого репозитория для распространения своих пакетов. Простейший способ помещения пакета в открытый репозиторий предусматривает использование онлайнового мастера загрузки пакетов, который проведет через все шаги, необходимые для загрузки пакета.

Чтобы запустить онлайновый мастер загрузки пакетов, выберите пункт меню Upload Package (Загрузить пакет) на веб-сайте NuGet.org.

Использование инструмента командной строки NuGet

В качестве альтернативы для развертывания пакетов можно воспользоваться функциональностью публикации, встроенной в инструмент командной строки NuGet. Этому методу часто отдают предпочтение перед непосредственной работой с веб-сайтом NuGet.org, поскольку если этот инструмент командной строки применяется для генерации пакетов, то несложно запустить его еще раз для выполнения публикации этих пакетов. Команда очень проста: `nuget push [имя пакета]`.

Например, команда для публикации созданного ранее пакета `MyApplication` выглядит следующим образом:

```
nuget push MyApplication
```

При запуске этой команды в первый раз, скорее всего, возникнет ошибка, указывающая на то, что не был задан ключ API для источника пакета, который вы пытаетесь опубликовать (в открытом репозитории NuGet.org). Ключи API – это уникальные и секретные маркеры, создаваемые репозиторием, так что они могут контролировать доступ к репозиторию. Хотя открытый репозиторий NuGet.org позволяет публиковать пакеты кому угодно, имеющему учетную запись, эту учетную запись еще нужно получить. К счастью, NuGet.org автоматически генерирует ключ API при создании учетной записи. Для извлечения этого ключа просто войдите на сайт и посетите страницу своего профиля. На этой странице имеется раздел API Key (Ключ API), который содержит ссылку `click to show` (щелкните для отображения). Щелкните на этой ссылке и скопируйте свой ключ API.

После копирования ключа API понадобится сообщить о нем NuGet. Для этого введите команду `nuget setApiKey [Ключ API]`, например:

```
nuget setApiKey ae19257f-9f0c-4dcf-b46a-60792fd5ff2d
```

Имея ключ API, вы теперь должны быть способны выполнить команду `nuget push`, чтобы опубликовать свои пакеты в открытом репозитории пакетов NuGet.org.

Размещение собственного репозитория пакетов

Для потребления пакетов в своих проектах помещать их в открытый репозиторий NuGet.org не обязательно. На самом деле вам даже не придется покидать свою локальную машину!

Инструмент NuGet предлагает два основных способа для размещения и использования собственных пакетов: установка репозитория в файловой системе и размещение в своем экземпляре веб-сервера NuGet.

Использование репозитория в файловой системе

Репозиторий в файловой системе представляет собой коллекцию пакетов, хранящихся в файловой системе, к которой у вас есть доступ. Его очень легко настраивать и запускать.

Чтобы взглянуть, как работает репозиторий в файловой системе, выполните несколько простых шагов.

1. Начните с создания новой папки по имени C:\NuGetPackages на локальном жестком диске.
2. Затем откройте диалоговое окно настроек NuGet (через пункт меню Tools⇒Library Package Manager⇒Package Manager Settings (Сервис⇒Диспетчер библиотечных пакетов⇒Настройки диспетчера пакетов) в Visual Studio) и переключитесь на раздел Package Sources (Источники пакетов) для добавления нового источника пакетов.
3. Далее добавьте новый источник пакетов, указав его имя в поле Name (Имя) и путь в поле Source (Источник), которым в данном случае является C:\NuGetPackages. Результат добавления нового источника пакетов показан на рис. Б.2.

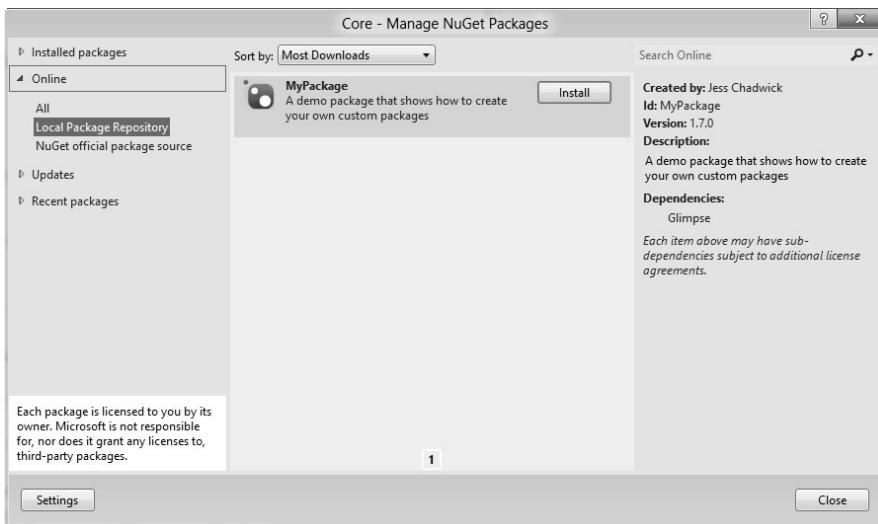


Рис. Б.2. Добавление нового источника пакетов

4. Наконец, щелкните на кнопке Add (Добавить) для добавления репозитория к списку источников.

При использовании диспетчера пакетов в следующий раз вы увидите в списке свой новый источник, при этом любые пакеты, добавляемые в данный источник (т.е. в папку C:\NuGetPackages), должны присутствовать в списке пакетов диспетчера, доступных для установки в проекте (рис. Б.3).

В этот момент у вас может возникнуть мысль о том, что создание репозитория пакетов на локальном жестком диске выглядит нелепо — и возможно вы правы. Тем не менее, имейте в виду, что путь к файлу не обязательно должен указывать на локальный жесткий диск. “Файловой системой”, которая хранит ваши пакеты, может быть любой совместно используемый ресурс, доступный через проводник файлов Windows, в том числе сетевые папки. В действительности размещение специального пакета NuGet вашей команды в централизованной сетевой папке является простым и эффективным способом обеспечения того, что все члены команды имеют доступ к тем же самим пакетам.

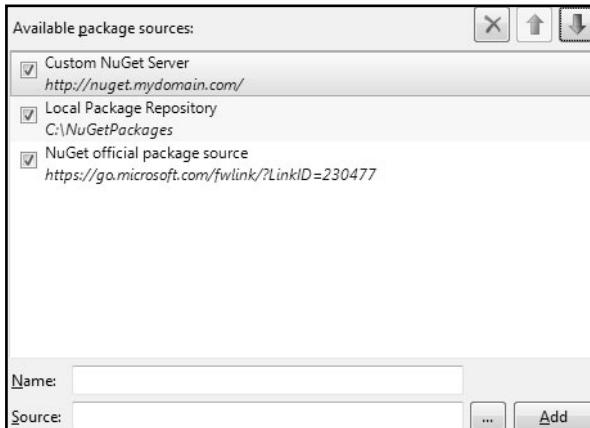


Рис. Б.3. Новый источник должен присутствовать в списке пакетов диспетчера

Размещение репозитория на сервере NuGet

Сервер NuGet Server представляет собой веб-сайт, который размещает набор веб-служб OData, содержащих информацию о пакетах NuGet; это тот же самый веб-сайт, который поддерживает открытый репозиторий пакетов NuGet.org. Хотя размещение репозитория пакетов в файловой системе обеспечивает простой путь для быстрой его установки и запуска, размещение его на собственном экземпляре NuGet Server в основном столь же просто, однако предоставляет намного большие возможности и гибкость. Ниже перечислены шаги по размещению собственного экземпляра NuGet Server.

1. Для начала откройте Visual Studio и создайте новый проект веб-приложения, используя шаблон ASP.NET Empty Web Application (Пустое веб-приложение ASP.NET).
2. С помощью диспетчера пакетов NuGet найдите и установите NuGet-пакет NuGet.Server, который загрузит и сконфигурирует все, что необходимо для запуска NuGet Server.
3. По умолчанию пакет NuGet.Server также создает папку Packages, которая действует в качестве стандартного местоположения для размещения ваших файлов пакетов NuGet. Если вы предпочитаете хранить пакеты NuGet где-то в другом месте, введите путь к новой папке Packages в подэлементе packagesPath элемента appSettings внутри файла web.config проекта.
4. Как только все настроено, веб-сайт можно развернуть как любой другой сайт. Имейте в виду, что сайт должен иметь возможность доступа к файлам в папке Packages, где бы она ни находилась.

После развертывания веб-сайта и проверки того, что все работает, наступает время для добавления нового сайта как еще одного источника пакетов в диалоговом окне настроек NuGet (открываемого через пункт меню Tools⇒Library Package Manager⇒Package Manager Settings в Visual Studio).

Данный шаг похож на добавление репозитория в файловой системе, показанное в предыдущем примере, за исключением того, что на этот раз значением поля Source будет URL экземпляра NuGet Server. Список доступных источников пакетов теперь должен выглядеть, как показано на рис. Б.4.

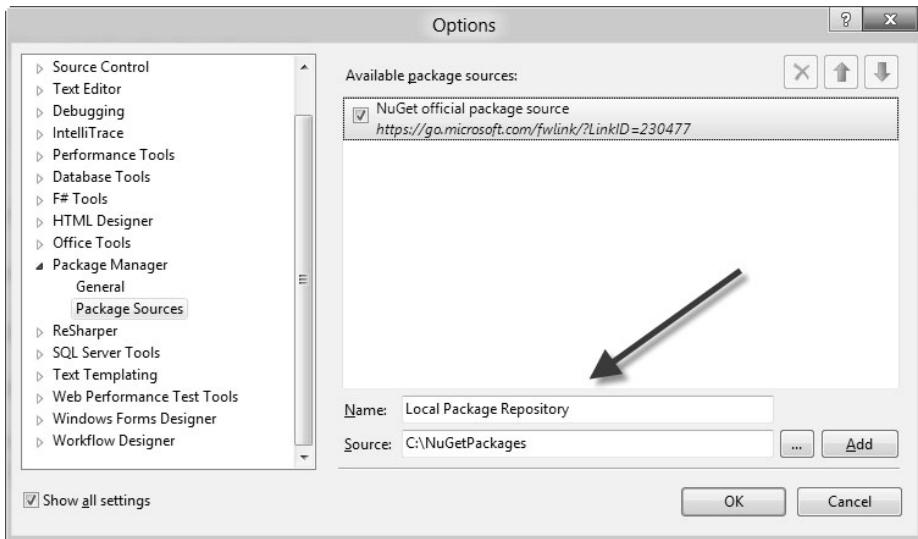


Рис. Б.4. Доступные источники пакетов

И снова при следующем открытии диспетчера пакетов вы увидите в списке новый источник, а все добавляемые в него пакеты будут отображаться в списке пакетов диспетчера, доступных для установки в проекте.

Советы, трюки и ловушки

С технической точки зрения использовать инструмент NuGet для загрузки, установки и конфигурирования зависимостей сборок, и даже для добавления контента в приложение, очень легко. Также довольно легко создавать и распространять собственные пакеты NuGet. Тем не менее, существуют несколько сценариев, в которых NuGet может вызывать некоторые сложности, особенно при работе с крупными приложениями с множеством проектов и/или несколькими командами разработчиков. В следующем списке представлены некоторые ловушки, на которые следует обращать особое внимание при работе с NuGet, а также ряд советов и трюков, помогающих извлечь максимум из NuGet в своих проектах.

Ловушка: NuGet не решает проблемы “ада DLL”

Фраза “ад DLL” (или более общая – “ад зависимостей”) отражает сложности, возникающие в ситуации, когда приложения имеют зависимости, такие как сборки или другие библиотеки, ссылка на которые производится динамически во время выполнения. Хотя и верно то, что одной из главных целей инструмента NuGet является обеспечение того, что все сборки актуальны и не конфликтуют друг с другом – и в основном для этого делается немало – не всегда возможно обеспечить выполнение данного обещания.

Наиболее распространенный конфликт возникает, когда два пакета ожидают совершенно разных версий третьего пакета. Например, Ninject и Glimpse разделяют зависимость от NLog; однако Ninject зависит от *максимальной* версии NLog, которой является 2.0, а Glimpse – от *минимальной* версии NLog, представляющей собой 2.5. Очевидно, что эти два пакета не могут быть включены в один и тот же проект! К счас-

тью, такой конфликт случается на очень раннем этапе и NuGet делает его легко обнаруживаемым (NuGet отобразит сообщение об ошибке и откажется добавлять пакеты), поэтому его влияние может быть минимизировано.

Гораздо худший конфликт может возникнуть, когда два пакета зависят от разных версий третьего пакета – как в предыдущем случае – но их метаданные NuSpec не предоставляют достаточной информации, чтобы инструмент NuGet мог знать, что эти два пакета конфликтуют друг с другом. Другими словами, логика, которую NuGet применяет для определения, какие версии сборок принадлежат вашему приложению, хороша только тогда, когда хороша информация, которую NuGet читает из файлов спецификации, поэтому если пакеты не предоставляют аккуратного и полного описания своих зависимостей, открываются возможности для возникновения непредусмотренных конфликтов.

Вспомните предыдущий пример, в котором пакет Ninject зависел от NLog v2.0, а пакет Glimpse – от NLog v2.5. Предположим, что в спецификациях для пакета Ninject забыли упомянуть, что он работает только с NLog v2.0 и предшествующими версиями, но не поддерживает NLog v2.5. Без этой информации NuGet будет устанавливать несовместимую версию NLog v2.5 – и сборки Ninject откажутся работать во время выполнения.

Это всего лишь несколько примеров классических конфликтов между версиями сборок, которые остаются проблемой даже в случае использования таких инструментов, как NuGet. При наличии надлежащих сведений о версиях и зависимостях, инструмент NuGet очень эффективен в попытках избежать потенциальных проблем совместимости, но ничто не защищено от неумелого использования – вы по-прежнему должны соблюдать осторожность, когда смешиваете разнообразные сборки!

Совет: используйте команду `Install-Package -Version` для установки специфичной версии пакета

Могут быть случаи, когда необходимо установить или обновить пакет, но избежать установки самой последней версии этого пакета.

Например, предположим, что приложение в текущий момент использует Ninject v2.1.0 и желательно произвести обновление до последней версии (скажем, v2.2.5), имеющей новые возможности, которые планируется задействовать. Проблема в том, что вы также обнаружили наличие в *последней версии* (к примеру, v2.4.0) известной ошибки, которая нарушит работу вашего приложения.

В этом сценарии, если вы запросите пакет Ninject через графический пользовательский интерфейс *Manage NuGet Packages* (Управление пакетами NuGet) или посредством команды `Install-Package` в консоли диспетчера команд, инструмент NuGet определит, что последней версией является v2.4.0 и предположит, что именно ее вы хотите установить. Конечно же, вам известно другое.

К счастью, NuGet поддерживает флаг `-Version`: механизм, который позволяет вам, как разработчику, переопределять интеллектуальную обработку версий NuGet и указывать точную версию пакета, которую необходимо установить (хотя эта опция и не доступна в графическом пользовательском интерфейсе). Для использования этой опции ее потребуется добавить в конец команды `Install-Package`, вводимой в консоли диспетчера команд.

Например, с помощью следующей команды можно было бы установить пакет Ninject версии 2.2.5, избегая обновления до версии 2.4.0:

```
Install-Package Ninject -Version 2.2.5
```

Эта команда загрузит и установит Ninject v2.2.5 невзирая на то, что NuGet самостоятельно вычисляет самую последнюю версию пакета Ninject.

Совет: используйте систему Semantic Versioning

Независимо от того, заботитесь ли вы о номере версии сборки, все сборки должны его иметь. Более того, чтобы создавать и распространять пакеты NuGet, потребуется назначать им уникальные номера версий, которые не только отличают их от других выпусков того же самого пакета, но также указывают порядок появления выпусков. По умолчанию Visual Studio генерирует такие номера версий, но когда вы начинаете создавать собственные пакеты NuGet, номера версий становятся более значимыми и вам, скорее всего, захочется играть более активную роль в определении и управлении ими.

По самой своей природе версии продуктов и сборок в лучшем случае относительны, а в худшем – произвольны. Например, “версия 1.0” одного продукта обычно не имеет ничего общего с “версией 1.0” другого продукта. В сущности, что означает “версия 1.0”? Чем она отличается от версий “0.1”, “1.5” или “2.0” того же самого продукта?

Поскольку назначение версий пакетам является, вероятно, самым важным аспектом управления зависимостями, в NuGet применяется популярная схема управления версиями под названием Semantic Versioning (<http://semver.org>). В то время как спецификация Semantic Versioning основана на ряде детальных правил, которые определяют различные части номеров версий, семантические версии в действительности сводятся к шаблону [Старший номер] . [Младший номер] . [Номер обновления]. Каждая часть в семантической версии представляет собой неотрицательное целое число, которое начинается с 0 и увеличивается на 1 при каждом изменении кодовой базы, которого достаточно для обеспечения смены версии.

Что в этом случае означает утверждение “изменение кодовой базы, которого достаточно для обеспечения смены версии”? Вообще говоря, части семантической версии увеличиваются в перечисленных ниже случаях.

- **Старший номер.** Всегда, когда вводятся изменения, не поддерживающие обратной совместимости.
- **Младший номер.** Всегда, когда вводятся новые изменения, поддерживающие обратную совместимость.
- **Номер обновления.** Всегда, когда вводятся изменения, поддерживающие обратную совместимость, в существующее поведение.

К распространяемым NuGet-пакетам можно применять любую схему управления версиями, но NuGet будет использовать описанные выше правила к любым номерам версий для определения, когда обновления являются подходящими.

Совет: помечайте “бета-пакеты” с помощью маркеров предварительной версии

Что случится, если необходимо распространить определенным пользователям “тестовый” пакет, который еще не полностью готов к выпуску в место общего хранения? В большинстве случаев при выпуске этого пакета в репозиторий инструмент NuGet автоматически загрузит и установит его для всех пользователей, что не является желаемым результатом. К счастью, NuGet поддерживает концепцию *пакетов предварительного выпуска* – пакетов, которые существуют бок о бок в том же самом репозитории с пакетами нормального “выпуска”, но скрыты от пользователей до тех пор, пока они специально их не запросят.

Чтобы пометить пакет как имеющий предварительный выпуск, преобразуйте номер версии пакета в номер версии предварительного выпуска Semantic Versioning, поместив после главного номера версии дефис (-) и затем любой алфавитно-цифровое обозначение версии предварительного выпуска. Например, номер версии “1.0-beta” указывает, что пакет имеет предварительный выпуск для “версии 1.0”, которая появится в какой-то момент в будущем.

Затем разверните пакет предварительного выпуска в том же репозитории, что и нормальные пакеты, где он будет доступен NuGet, но не виден пользователям, выполняющим обычные действия по управлению пакетами. Для потребления пакетов предварительного выпуска добавьте флаг `-Prerelease` к любой команде запроса или установки NuGet. Например, если для версии пакета `MyApplication` был подготовлен предварительный выпуск “1.0-beta”, то команда `Get-Package` без флага `-Prerelease` не сможет найти этот пакет:

```
PM> Get-Package MyApplication
```

Тем не менее, добавление флага `-Prerelease` позволяет обнаружить в том же репозитории бета-пакет:

```
PM> Get-Package -ListAvailable -Filter -Prerelease MyApplication
```

Id	Version	Description/Release Notes
--	-----	-----
MyApplication	1.0-beta	My awesome package.

Пакет предварительного выпуска можно затем установить с помощью стандартной команды `Install-Package`, в которой применяется флаг `-Prerelease`:

```
PM> Install-Package -Prerelease MyApplication
```

Наконец, когда тестирование завершено и пакет готов к выпуску, суффикс “-beta” может быть отброшен; версия пакета “1.0” будет видимой любому и, начиная с этого момента, пакет будет включен как возможный кандидат на установку во все операции с пакетами NuGet.



Поддержка версий предварительных выпусков определена как часть схемы Semantic Versioning, так что этот совет в действительности является расширением предыдущего совета относительно Semantic Versioning. Это еще одна причина применять систему Semantic Versioning к своим пакетам!

Ловушка: избегайте указания “строгих” зависимостей от версий в файлах NuSpec

Конфигурационные файлы NuSpec позволяют пакетам указывать другие пакеты, а также версии этих пакетов, от которых они зависят. Раздел зависимостей файла NuSpec должен быть максимально подробным и детальным, чтобы инструмент NuGet имел достаточно информации для принятия корректных решений о том, какие пакеты могут конфликтовать друг с другом, а какие версии пакетов могут быть безопасно установлены.

Как объяснялось ранее при упоминании “ада DLL”, с разработкой программного обеспечения связан один непреложный факт, заключающийся в том, что иногда версии различных сборок просто не могут смешиваться. В результате приложение может стать непригодным для использования. Таким образом, когда распространяемый пакет имеет известные проблемы при взаимодействии с последними версиями библи-

отеки, которая от него зависит, в общем случае рекомендуется очень четко описать этот конфликт в конфигурационном файле NuSpec для пакета.

Например, пакет Ninject в примере с “адом DLL” был совместимым только с версиями NLog, предшествующими v2.0, и имел известные проблемы с версией NLog v2.5. Его файл NuSpec мог бы содержать следующую строку, которая указывает, что пакет ожидает как *максимум* версию v2.0 пакета NLog и что инструмент NuGet не должен устанавливать пакет NLog с версией, превышающей указанную:

```
<dependency id="NLog" version="2.0" />
```

Хотя подобного рода спецификации конкретных версий могут быть необходимы для устранения конфликтов во время выполнения, имейте в виду, что они косвенно влияют на все пакеты в проекте и накладываемые ими ограничения могут спровоцировать множество конфликтов, существенно затрудняя управления пакетами. На самом деле, хуже всего то, что такие конфликты часто применяются при обновлении до последних версий того же самого пакета, в котором специфичная зависимость от версии была разрешена.

Зависимость от конкретной версии сборки иногда действительно неизбежна, однако очень хорошо подумайте, прежде чем вводить такое ограничение, т.к. это решение потребует больших затрат времени в случае отмены в будущем.

Если вы когда-нибудь столкнетесь с ситуацией, когда необходимо обновить пакет от конкретной версии до более высокой, но не конкретной его версии, зайдите в консоль управления пакетами и воспользуйтесь командой `Install-Package` с переключателем `-IgnoreDependencies` для аннулирования любых конфликтов, о которых сообщает NuGet. Например:

```
Install-Package -IgnoreDependencies Ninject
```

Эта команда укажет инструменту NuGet на то, что он должен установить последнюю версию пакета и игнорировать любые конфликты зависимостей, которые он может обнаружить.



Помните, что при использовании переключателя `-IgnoreDependencies` вы обходите все защитные средства NuGet, поэтому ответственность за правильность всех зависимостей между пакетами возлагается на вас!

Совет: используйте специальные репозитории для управления версиями пакетов

Инструмент NuGet имеет один заранее сконфигурированный репозиторий: открытый репозиторий, размещенный на веб-сайте NuGet.org. Это удобно для большинства разработчиков, учитывая, что открытый репозиторий NuGet.org, скорее всего, будет единственным необходимым аспектом. Недостаток работы с открытым репозиторием NuGet заключается в том, что вы и ваша организация не имеют никакого контроля над пакетами, размещаемыми в этом репозитории.

Хотя стабильный цикл выпуска пакетов с постоянными обновлениями и увеличивающимися версиями обычно приветствуется, некоторые команды могут испытывать неудобства из-за частых изменений и отсутствия контроля над происходящим. Например, если вы используете открытый репозиторий NuGet в качестве главного источника пакетов, а команда разработчиков Ninject выпустит новую версию, несовместимую с вашим приложением, то NuGet попытается получить обновление до последней версии пакета Ninject, даже когда известно о несовместимостях с вашим приложением.

В этом случае можно легко проигнорировать рекомендации по обновлению NuGet, но ситуация становится несколько сложнее, когда выходят обновления для других пакетов, применяемых в приложении, причем эти обновления зависят от последней версии Ninject – инструмент NuGet также откажется обновлять эти пакеты без получения последней версии Ninject.

Один из способов избежать таких “вынужденных” обновлений заключается в восстановлении контроля над обновлениями пакетов, заменяя стандартный открытый репозиторий NuGet собственным специальным репозиторием. После этого специальный репозиторий можно заполнить только пакетами, которые вы и ваша команда считаете “приемлемыми”.

Пакеты могут поступать из произвольных мест. Некоторые из них могут быть специальными пакетами, созданными вами, но большинство, скорее всего, окажется пакетами, которые вы извлекаете прямо из открытого репозитория NuGet. Замена стандартного репозитория NuGet собственным репозиторием позволяет получить лучшее из обоих миров. Это даст возможность пользоваться мощными средствами управления зависимостями, но по-прежнему сохранять полный контроль над пакетами, доступными для установки.

Действительно, многих разработчиков может не интересовать выбор конкретных пакетов, к которым инструмент NuGet имеет доступ, но в некоторых сценариях – особенно, когда вовлечено множество команд – создание стабильного пополняемого репозитория помогает организовывать управляемые среды для минимизации влияния со стороны “ада DLL”.

Совет: сконфигурируйте свои построения непрерывной интеграции для генерации пакетов NuGet

Построения непрерывной интеграции и NuGet – это “брак, заключенный на небесах”! Хотя для разных платформ непрерывной интеграции специфика будет отличаться, общий процесс создания пакетов NuGet из построений непрерывной интеграции выглядит одинаково.

1. Создайте файлы NuSpec для всех пакетов NuGet, которые необходимо сгенерировать, и не забудьте проверить их в системе управления исходным кодом.
2. Сконфигурируйте построения для запуска инструмента командной строки NuGet в конце каждого успешно завершившегося построения.
3. Обеспечьте передачу уникальной версии пакета каждый раз, когда генерируется новый набор пакетов. На самом деле, большинство систем непрерывной интеграции включают уникальный номер построения, которым можно воспользоваться.
4. Наконец, заставьте сервер построения переместить сгенерированные пакеты в центральный репозиторий.

Дополнительные сведения по настройке специального репозитория NuGet были приведены в разделе “Размещение собственного репозитория пакетов” ранее в этом приложении.

Генерация пакетов NuGet как часть построения непрерывной интеграции может оказаться исключительно ценным дополнением к процессу непрерывной интеграции и развертывания.

Резюме

Хотя управление ссылками на сборки временами может оказаться довольно трудным, NuGet представляет собой мощную систему управления зависимостями, которая помогает устраниТЬ большую часть сложностей. Пакеты NuGet невероятно легко потреблять, и они несложны в создании и распространении. При разумном использовании NuGet может быть следующим самым мощным инструментом в комплекте разработки сразу же после собственно среды Visual Studio.

Рекомендуемые приемы

В этой книге было раскрыто множество тем с различной степенью детализации и предложено большое число полезных советов. Тем не менее, иногда довольно трудно судить о том, насколько важной является любая заданная порция информации в контексте длинного технического описания.

В этом приложении собраны многие рекомендуемые приемы, что позволит вам быстро выяснить, следете ли вы популярным и рекомендуемым шаблонам и приемам, изложенным в настоящей книге.

Используйте для управления зависимостями диспетчер пакетов NuGet

Диспетчер пакетов NuGet является полезным средством как для одиночных разработчиков, так и для команд. Вместо того чтобы тратить массу времени на выполнение проверок, не выпущены ли новые версии проектов, от которых зависит ваше приложение, позвольте инструменту NuGet все это обрабатывать самостоятельно!

Если в организации имеется несколько команд, разделяющих общие библиотеки, рассмотрите возможность создания специальных пакетов NuGet для таких совместно используемых библиотек и размещения специального репозитория NuGet для обеспечения более эффективного распространения и управления версиями.

Полагайтесь на абстракции

Абстракции способствуют слабому связыванию систем со значительным разделением контрактов и реализации. Абстракции легко менять местами, что не только обеспечивает более простое сопровождение, но также и критически важно для модульного тестирования.

Избегайте использования ключевого слова new

Каждый раз, когда вы применяете ключевое слово `new` для создания нового экземпляра конкретного типа, вы по определению не полагаетесь на абстракцию. Хотя часто это не является проблемой (например, `new StringBuilder()`, `new List<string>()` и т.д.), воспользуйтесь моментом и подумайте, не лучше ли выразить создаваемый объект в виде зависимости, предназначенной для внедрения. Всегда, когда возможно, позволяйте создавать объект другому компоненту!

Избегайте прямых ссылок на объект `HttpContext` (используйте `HttpContextBase`)

В ASP.NET MVC (и позже в .NET Framework 4) был введен `System.Web.Abstractions` — набор абстракций для множества ключевых частей ASP.NET Framework. Приведенный ранее совет “Полагайтесь на абстракции” распространяется также и на эти классы. В частности, одним из наиболее широко ссылаемых объектов при разработке ASP.NET является `HttpContext` — вместо него отдавайте предпочтение применению абстракции `HttpContextBase`.

Избегайте “магических строк”

“Магические строки” — важные, однако произвольные строковые значения — могут быть удобны, а во многих ситуациях и обязательны; тем не менее, с ними связано много проблем. Ниже перечислены некоторые из наиболее крупных проблем с “магическими строками”:

- они не имеют никакого внутреннего смысла (например, трудно сказать, как один “идентификатор” связан с другим “идентификатором”, да и связан ли вообще);
- легко разрушаются из-за неточного написания или неправильного регистра символов;
- не особенно хорошо поддаются рефакторингу;
- способствуют повсеместному дублированию.

Рассмотрим два примера, в первом из которых используются “магические строки” для доступа к данным в словаре `ViewData`, а во втором примере, после рефакторинга, обращение к тем же данным производится с помощью строго типизированной модели:

```
<p>
    <label for="FirstName">First Name:</label>
    <span id="FirstName">@ViewData["FirstName"]</span>
</p>

<p>
    <label for="FirstName">First Name:</label>
    <span id="FirstName">@Model.FirstName</span>
</p>
```

“Магические строки” привлекают простотой применения, когда они только определяются, но эта простота часто об оборачивается проблемами позже, когда дело доходит до сопровождения.

Отдавайте предпочтение моделям перед словарем `ViewData`

Как предполагалось в предыдущем примере, словарь `ViewData` является одним из самых привлекательных мест для использования “магических строк” в приложении ASP.NET MVC. Тем не менее, применения этого словаря лучше избегать. Удобным инструментом, который позволит отказаться от присваивания и извлечения данных напрямую из словаря `ViewData`, могут служить строго типизированные модели.

Не записывайте HTML-разметку в “серверный” код

Следуйте приему разделения ответственности: визуализация HTML-разметки не входит в обязанности контроллеров и другого “серверного” кода. Исключениями, конечно же, являются вспомогательные методы и классы пользовательского интерфейса, единственная работа которых заключается в помощи представлениям визуализировать код. Эти классы должны рассматриваться как часть представления, а не “серверных” классов.

Не выполняйте бизнес-логику в представлениях

Также справедлива и противоположность предыдущего рекомендуемого приема: представления не должны содержать бизнес-логику. На самом деле представления вообще должны содержать как можно меньше логики! Представления должны быть сосредоточены на том, как отображать предоставляемые данные, и не выполнять никаких действий над этими данными.

Консолидируйте часто используемые фрагменты представлений с помощью вспомогательных методов

Понятия “пользовательские элементы управления”, “серверные элементы управления” и “элементы управления”, в общем, весьма широко распространены, и не зря. Эти концепции помогают консолидировать часто применяемый код и логику в центральном местоположении, упрощая их повторное использование и сопровождение. Однако инфраструктура ASP.NET MVC не основана на элементах управления — вместо этого она полагается на парадигму “вспомогательных методов”, при которой методы делают работу, выполняемую ранее элементами управления. Это может касаться целого раздела HTML-разметки (то, что мы привыкли называть “элементом управления”) или чего-то более простого, такого как строго типизированный доступ к часто обращаемому URL. Например, вы могли заметить множество одинаковых ссылок на страницу Membership (`~/membership`):

```
@Html.ActionLink("Membership", "Index", "Membership", [...])
```

Этот вызов можно консолидировать (и устраниТЬ “магические строки”), превратив его во вспомогательный метод:

```
@Html.MembershipLink()
```

Отдавайте предпочтение презентационным моделям перед прямым использованием бизнес-объектов

В целом пытайтесь избегать разрешения изменениям, вносимым в бизнес-модель, напрямую влиять на представление. В этом помогают презентационные модели.

Инкапсулируйте операторы `if` во вспомогательные методы HTML в представлениях

Интеграция кода и разметки является довольно мощной, однако она также может стать и весьма запутанной. Взгляните на следующий (относительно простой) оператор `if/else`:

```
@if(Model.IsAnonymousUser) {  
      
} else if(Model.IsAdministrator) {  
      
} else if(Model.Membership == Membership.Standard) {  
      
} else if(Model.Membership == Membership.Preferred) {  
      
}
```

Этот довольно неясный код визуализирует в точности ту же самую разметку, исключая одну часть (URL). Ниже продемонстрирован другой подход:

```
public static string UserAvatar(this HtmlHelper<User> helper)  
{  
    var user = helper.ViewData.Model;  
    string avatarFilename = "anonymous.jpg";  
  
    if (user.IsAnonymousUser)  
    {  
        avatarFilename = "anonymous.jpg";  
    }  
    else if (user.IsAdministrator)  
    {  
        avatarFilename = "administrator.jpg";  
    }  
    else if (user.Membership == Membership.Standard)  
    {  
        avatarFilename = "member.jpg";  
    }  
    else if (user.Membership == Membership.Preferred)  
    {  
        avatarFilename = "preferred_member.jpg";  
    }  
  
    var urlHelper = new UrlHelper(helper.ViewContext.RequestContext);  
    var contentPath = string.Format("~/content/images/{0}", avatarFilename);  
    string imageUrl = urlHelper.Content(contentPath);  
    return string.Format("<img src='{0}' />", imageUrl);  
}
```

Теперь можно просто вызывать этот вспомогательный метод везде, где требуется аватар пользователя:

```
@Html.UserAvatar()
```

Такой подход не только яснее, но также является и более декларативным, к тому же он перемещает показанную логику в центральное местоположение, поэтому ее намного легче сопровождать. Например, если требования изменяются и сайт нуждается в поддержке специальных аватаров, то вспомогательный метод `Html.UserAvatar()` может быть модифицирован в одном месте.

Отдавайте предпочтение явным именам представлений

В большинстве примеров кода для действий контроллеров ASP.NET MVC вызывается метод `View()` без указания имени представления. Это приемлемо для простого демонстрационного кода, но когда тесты или другие действия начинают вызывать

друг друга, ущерб от такого подхода становится очевидным. Если имя представления не указано, по умолчанию ASP.NET MVC Framework считает, что оно совпадает с именем действия, которое было первоначально вызвано. Таким образом, вызов действия Index в следующем примере приведет к тому, что инфраструктура будет пытаться найти представление по имени Index.cshtml, которое, скорее всего, не существует (однако определено существует представление List.cshtml):

```
public ActionResult Index()
{
    return List();
}
public ActionResult List()
{
    var employees = Employee.GetAll();
    return View(employees);
}
```

Если изменить действие List для вызова метода View() со специфичным именем представления (как показано ниже), то все нормально заработает:

```
public ActionResult List()
{
    var employees = Employee.GetAll();
    return View("List", employees);
}
```

Отдавайте предпочтение объектам параметров перед длинными списками параметров

Этот совет не является специальным для ASP.NET MVC – длинные списки параметров обычно считаются “признаком кода”, поэтому их следует избегать всегда, когда это возможно. Вдобавок мощные связыватели моделей ASP.NET MVC существенно упрощают следование этому совету. Взгляните на показанные ниже два противоположных фрагмента, в первом из которых используется длинный список параметров:

```
public ActionResult Create(
    string firstName, string lastName, DateTime? birthday,
    string addressLine1, string addressLine2,
    string city, string region, string regionCode, string country
    [... и многие, многие другие]
)
{
    var employee = new Employee( [Длинный список параметров...] )
    employee.Save();
    return View("Details", employee);
}
```

а во втором – объект параметра:

```
public ActionResult Create(Employee employee)
{
    employee.Save();
    return View("Details", employee);
}
```

Пример с объектом параметра намного более прямолинеен, и они использует мощные связыватели моделей и проверку достоверности моделей ASP.NET MVC Framework, делая такой код намного безопаснее и проще в сопровождении.

Инкапсулируйте разделяемую/общую функциональность, логику и данные с помощью фильтров действий или дочерних действий (`Html.RenderAction`)

Каждый веб-сайт любой значительной сложности будет иметь общие элементы, которые появляются на множестве (или, возможно, на всех) страниц в приложении. Каноническим примером такого типа глобально применяемой логики и контента является основное меню навигации по сайту, которое присутствует на каждой странице сайта. Данные для таких общих элементов должны откуда-то поступать, но явное извлечение данных в каждом действии контроллера может привести к сложностям при сопровождении. Фильтры действий и/или дочерние действия (выполняемые через метод `Html.RenderAction()`) обеспечивают центральное местоположение для хранения логики такого типа.

Взгляните на следующий фрагмент разметки (вырезанный из более крупной страницы разметки), который визуализирует навигационные элементы в списке:

```
<ul id="global-menu">
    @foreach (var menuItem in ViewData.SingleOrDefault<NavigationMenu>())
    {
        <li class="@menuItem.IsSelected ? "selected" : null">
            @Html.RouteLink(menuItem.DisplayName, menuItem.RouteData)
        </li>
    }
</ul>
```

Объект `ViewData` типа `NavigationMenu` должен откуда-то поступать. Поскольку он может быть сконфигурирован на выполнение перед каждым запросом контроллера, фильтры действий становятся отличным кандидатом на заполнение `ViewData` глобально требуемыми данными подобного рода. Ниже представлен фильтр действия, который заполняет данные меню навигации, требуемые в предыдущем примере:

```
public class NavigationMenuPopulationFilter : ActionFilterAttribute
{
    private readonly INavigationDataSource _dataSource;
    public NavigationMenuPopulationFilter(INavigationDataSource dataSource)
    {
        _dataSource = dataSource;
    }
    public override void OnActionExecuting(ActionExecutingContext filterContext)
    {
        NavigationMenu mainMenu = _dataSource.GetNavigationMenu("main-menu");
        filterContext.Controller.ViewData["MainNavigationMenu"] = mainMenu;
    }
}
```

Этот фильтр очень прост – он получает корректную модель данных меню навигации из некоторого источника данных и добавляет ее в коллекцию `ViewData` до выполнения запрошенного действия. С этой точки зрения, любой компонент, которому требуется меню навигации, может извлечь его из `ViewData`.

Отдавайте предпочтение группированию действий в контроллеры на основе того, как они связаны с бизнес-концепциями

Например, рассмотрите вопрос создания контроллера `CustomersController` для хранения действий, относящихся к работе с заказчиками.

Избегайте группирования действий в контроллеры на основе технических отношений

Например, избегайте создания контроллера `AjaxController`, который бы содержал все действия AJAX, открываемые сайтом. Вместо этого группируйте эти действия по связанным концепциям (скажем, действия AJAX, которые предоставляют данные о заказчиках или частичные представления, должны находиться в контроллере `CustomersController` вместе со всеми другими действиями, относящимися к работе с заказчиками).

Отдавайте предпочтение фильтрам действий на самом высоком подходящем уровне

Большинство атрибутов фильтров действий могут применяться на уровне либо метода (действие), либо класса (контроллер). Когда атрибут применяется ко всем действиям в контроллере, отдавайте предпочтение размещению этого атрибута на самом контроллере, а не на каждом индивидуальном классе. Также подумайте, не будет ли атрибут больше уместен в цепочке зависимостей контроллера (т.е. в одном из его базовых классов).

Отдавайте предпочтение нескольким представлениям (и/или частичным представлениям) перед сложной логикой `if-then-else`, которая отображает и скрывает разделы

Шаблон Web Forms Page Controller (Контроллер страницы Web Forms) способствует обратной отправке на одну и ту же страницу, возможно, отображая или скрывая определенные разделы на странице в зависимости от запроса.

Благодаря разделению ответственности ASP.NET MVC, этого часто можно избежать путем создания отдельных представлений для каждой из таких ситуаций, снижая или вообще устранивая потребность в сложной логике представления. Взгляните на следующий пример из `Wizard.cshtml`:

```
@if(Model.WizardStep == WizardStep.First) {  
    <!-- Первый шаг мастера -->  
} else if(Model.WizardStep == WizardStep.Second) {  
    <!-- Второй шаг мастера -->  
} else if(Model.WizardStep == WizardStep.Third) {  
    <!-- Третий шаг мастера -->  
}
```

Здесь представление само определяет, какой шаг мастера отображать, что опасно близко к бизнес-логике!

Давайте перенесем эту логику в контроллер (`WizardController.cs`), которому она принадлежит:

```
public ActionResult Step(WizardStep currentStep)
{
    // Это простая логика, но она могла быть НАМНОГО более сложной!
    string view = currentStep.ToString();
    return View(view);
}
```

и разделим исходное представление на несколько представлений, например, `First.cshtml`:

```
<!-- Первый шаг мастера -->
```

`Second.cshtml`:

```
<!-- Второй шаг мастера -->
```

`Third.cshtml`:

```
<!-- Третий шаг мастера -->
```

Отдавайте предпочтение шаблону Post-Redirect-Get при отправке данных формы

Шаблон проектирования Post-Redirect-Get (PRG) часто используется веб-разработчиками во избежание определенных дублированных отправок форм и позволяет пользовательским агентам вести себя интуитивно понятным образом в отношении закладок и кнопки обновления. Поскольку шаблон Web Forms Page Controller обычно заставляет разработчиков выполнять обратную отправку той же самой странице для всех действий в отдельном контексте (например, отображение данных о сотруднике, которые можно отредактировать и повторно отправить), шаблон PRG применяется в средах Web Forms не особенно часто. Тем не менее, поскольку инфраструктура ASP.NET MVC разделяет действия по отдельным URL, довольно часто могут возникнуть затруднения со сценариями обновления. Взгляните на следующую реализацию контроллера `EmployeeController`:

```
public class EmployeeController : Controller
{
    public ActionResult Edit(int id)
    {
        var employee = Employee.Get(id);
        return View("Edit", employee);
    }

    [AcceptVerbs(HttpVerbs.Post)]
    public ActionResult Update(int id)
    {
        var employee = Employee.Get(id);
        UpdateModel(employee);
        return View("Edit", id);
    }
}
```

В этом примере, когда пользователь осуществляет отправку в адрес действия `Update`, несмотря на то, что отобразится ожидаемое представление `Edit`, результирующим URL в браузере будет `/employees/update/1`. Если пользователь обновляет

страницу, помещает этот URL в список закладок и т.д., то последующие посещения будут обновлять информацию о сотруднике снова или возможно вообще не зарабатывают. В действительности от действия `Update` требуется обновление информации о сотруднике и затем перенаправление пользователя обратно на страницу `Edit` (Редактирование), обеспечивая возврат к исходному местоположению, где производилось редактирование. При таком сценарии шаблон PRG может быть применен следующим образом (первая часть файла не показана, приведен лишь раздел с измененным действием `Update`):

```
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Update(int id)
{
    var employee = Employee.Get(id);
    UpdateModel(employee);
    return RedirectToAction("Edit", new { id });
}
```

Несмотря на столь незначительное изменение, переход к использованию метода `RedirectToAction()` вместо `View()` обеспечит перенаправление клиентской стороны (в противоположность перенаправлению серверной стороны в исходном примере) после завершения методом `Update()` обновления информации о сотруднике, перемещая пользователя на правильный URL: `/employees/edit/1`.

Отдавайте предпочтение задачам запуска перед логикой, размещаемой в методе `Application_Start() (Global.asax)`

В большинстве демонстрационных приложений ASP.NET MVC будет даваться совет модифицировать метод `Application_Start()` в файле `Global.asax` для добавления логики, которая выполняется при запуске приложения. Хотя это определенно простейший и наиболее прямолинейный подход, инфраструктура `WebActivator` предлагает альтернативную концепцию задач запуска. Такие задачи легко реализовать, к тому же они автоматически обнаруживаются и выполняются во время запуска приложения. Эти задачи помогают обеспечить более чистый код и способствуют надлежащему соблюдению принципа единственной ответственности (Single Responsibility Principle), упомянутого в главе 5.

Отдавайте предпочтение атрибуту авторизации перед императивными проверками безопасности

Традиционно управление авторизацией выглядит следующим образом:

```
public ActionResult Details(int id)
{
    if (!User.IsInRole("EmployeeViewer"))
        return new HttpUnauthorizedResult();
    // Логика действия.
}
```

Это императивный подход, который затрудняет реализацию изменений на уровне приложения. Атрибут авторизации (`AuthorizeAttribute`) в ASP.NET MVC предлагает простой и декларативный способ авторизации доступа к действиям.

Показанный выше код может быть переписан так:

```
[Authorize(Roles = "EmployeeViewer")]
public ActionResult Details(int id)
{
    // Логика действия.
}
```

Отдавайте предпочтение использованию атрибута маршрута перед более обобщенными глобальными маршрутами

Разумеется, наиболее специфичным маршрутом является тот, который отображается напрямую на одно и только одно действие.

Подумайте об использовании маркера противодействия атакам CSRF

Для отправок форм, при которых важно соблюдение безопасности, инфраструктура ASP.NET MVC поддерживает меры по предотвращению некоторых видов распространенных атак. Одной из таких мер является маркер противодействия атакам CSRF. Этот маркер имеет компоненты как серверной, так и клиентской стороны. Следующий код вставляет специфичный для пользователя маркер в скрытое поле формы:

```
@using(Html.Form("Update", "Employee"))
{
    @Html.AntiForgeryToken()
    <!-- Остальная часть формы -->
}
```

А приведенный ниже код проверяет этот маркер на стороне сервера перед выполнением любой дальнейшей обработки отправленных данных:

```
[ValidateAntiForgeryToken]
[AcceptVerbs(HttpVerbs.Post | HttpVerbs.Put)]
public ActionResult Update(int id)
{
    // Обработать проверенную отправку формы.
}
```

Подумайте об использовании атрибута AcceptVerbs для ограничения способов вызова действий

Многие действия опираются на ряд предположений о том, как и когда они будут вызваны в контексте приложения. Например, такое предположение может заключаться в том, что действие Employee.Update будет вызываться из определенного вида страницы редактирования сведений о сотруднике, которая содержит форму со свойствами сотрудника, предназначеннную для отправки действию Employee.Update с целью обновления записи о сотруднике. Если это действие вызывается незапланированным способом (например, через запрос GET без отправки формы), скорее всего, оно работать не будет, и на самом деле может даже привести к непредвиденным проблемам.

Инфраструктура ASP.NET MVC Framework предлагает атрибут `AcceptVerbs`, помогающий ограничивать вызовы действий специфическими HTTP-методами.

Таким образом, упомянутый выше сценарий с `Employee.Update` можно реализовать следующим образом:

```
[AcceptVerbs(HttpVerbs.Post | HttpVerbs.Put)]
public ActionResult Update(int id)
```

Применение атрибута `AcceptVerbs` таким способом ограничит запросы к этому действию HTTP-методами POST или PUT. Все остальные запросы (например, GET) будут игнорироваться.

Подумайте об использовании кеширования вывода

Кеширование вывода является одним из самых простых путей получения дополнительной производительности от веб-приложения. Кеширование визуализированной HTML-разметки – это отличный способ уменьшения времени ответа, когда с момента предыдущего запроса контент не менялся либо изменялся мало. Для этих целей инфраструктура ASP.NET MVC Framework предлагает атрибут `OutputCacheAttribute`. Данный атрибут отражает функциональность кеширования вывода Web Forms и принимает многие из тех же самых свойств.

Подумайте об удалении неиспользуемых механизмов представлений

Инфраструктура ASP.NET MVC по умолчанию регистрирует механизмы представлений Web Forms и Razor, а это означает, что локатор представлений будет производить поиск в местоположениях представлений как Web Forms, так и Razor. В результате становится возможным использование в приложении любого или обоих типов представлений.

Тем не менее, ради согласованности большинство команд выбирают один тип представлений и применяют его повсеместно в приложении, делая поиск неиспользуемых представлений локатором представлений ASP.NET MVC несколько избыточным. К примеру, если в приложении решено применять только представления Razor, то локатор представлений продолжит поиск представлений Web Forms, несмотря на то, что ни одно из них не будет найдено.

К счастью, этих нежелательных накладных расходов можно избежать и слегка оптимизировать приложение, отменив регистрацию механизма представлений, который не используется.

В следующем примере показано, как отменить регистрацию механизма представлений Web Forms (оставив только механизм Razor):

```
var viewEngines = System.Web.Mvc.ViewEngines.Engines;
var webFormsEngine = viewEngines.OfType<WebFormViewEngine>()
    .FirstOrDefault();
if (webFormsEngine != null)
    viewEngines.Remove(webFormsEngine);
```

Просто выполните этот фрагмент кода во время фазы начального запуска приложения, и локатор представлений больше не будет тратить время на поиск несуществующих представлений.

Подумайте об использовании специальных результатов действий в уникальных сценариях

Конвейер запросов ASP.NET MVC поддерживает тщательно спланированное разделение ответственности, при котором каждый шаг в процессе завершает свою задачу и не делает ничего больше. Каждый шаг делает достаточно для того, чтобы снабдить последующие задачи исчерпывающей информацией о том, что им необходимо делать. Например, действие контроллера, которое решает, что для клиента должно быть визуализировано представление, не загружает механизм представлений и не требует от него выполнения представления. Оно просто возвращает объект `ViewResult` с информацией, которая нужна инфраструктуре для организации следующих шагов (скоро всего, этими шагами будет именно загрузка механизма представлений и выполнение самого представления).

Когда дело доходит до результатов действий контроллеров, в игру вступает декларативность. К примеру, инфраструктура ASP.NET MVC Framework предоставляет класс `HttpStatusCodeResult` со свойством `StatusCode`, но также определяет специальный класс `HttpStatusCodeResult` по имени `HttpUnauthorizedResult`. Хотя следующие две строки фактически делают то же самое, последняя обеспечивает более декларативное и строго типизированное выражение намерения контроллера:

```
return new HttpStatusCodeResult(HttpStatusCode.Unauthorized);  
return new HttpUnauthorizedResult();
```

Когда действия порождают результаты, которые не вписываются в “нормальные” результаты, подумайте о том, не окажется ли возврат специального результата действия более подходящим. Распространенные примеры включают такие вещи, как RSS-каналы, документы Word, электронные таблицы Excel и т.д.

Подумайте об использовании асинхронных контроллеров для задач, которые могут выполняться параллельно

Параллельное выполнение множества задач может предложить большие возможности по улучшению производительности вашего сайта. Для этого в ASP.NET MVC предусмотрен базовый класс `AsyncController`, который помогает облегчить обработку многопоточных запросов. Когда создается действие с логикой, интенсивно использующей процессор, посмотрите, есть ли в этом действии элементы, которые могут безопасно запускаться параллельно. За дополнительными сведениями по этому поводу обращайтесь в главу 11.

Перекрестные ссылки: целевые темы, функциональные возможности и сценарии

Ниже приведены списки концепций, которые рассматривались в этой книге, со ссылками на главы, в которых можно найти их упоминания.

Тема	Глава (главы)
Новые функциональные возможности ASP.NET MVC 4	Новые функциональные возможности ASP.NET MVC 4
Мобильные шаблоны	Глава 10
Пакетирование и минимизация JavaScript-кода	Глава 13
ASP.NET Web API	Глава 7
Асинхронные контроллеры	Глава 11
AllowAnonymousAttribute	Глава 9
Функциональные возможности ASP.NET MVC	Функциональные возможности ASP.NET MVC
Действия контроллеров	Глава 1
Фильтры действий	Глава 1
Маршрутизация	Главы 1, 14
Разметка Razor	Глава 1
Вспомогательные методы HTML	Главы 1, 3
Вспомогательные методы URL	Глава 1
Вспомогательные методы форм	Глава 3
Клиентская проверка достоверности	Главы 3, 8
Области	Глава 1

Тема	Глава (главы)
Результат JSON	Глава 6
Частичные представления	Главы 1, 6, 15,
Вспомогательные методы Razor	Глава 15
Привязка моделей	Глава 6
Проверка достоверности	Глава 3
Обработка ошибок	Глава 16
Механизмы представлений	Глава 1
Дочерние действия	Глава 12
Кеширование вывода	Глава 12
<code>bin_DeployableAssemblies</code>	Глава 19
Специальные шаблоны элементов	Глава 15
Типы проектов ASP.NET MVC	Типы проектов ASP.NET MVC
Empty (Пустой)	Глава 1
Internet Application (Интернет-приложение)	Глава 1
Intranet Application (Инtranет-приложение)	Глава 9
Mobile Application (Мобильное приложение)	Глава 10
Web API	Глава 6
Шаблоны и приемы	Шаблоны и приемы
Шаблон “модель-представление-контроллер” (MVC)	Глава 5
Многоуровневая модель	Глава 5
Принципы проектирования SOLID	Глава 5
Привязка и проверка достоверности моделей	Глава 6
Объектно-реляционное отображение (ORM)	Глава 8
Ведение журналов и мониторинг работоспособности	Глава 16
Модульное тестирование	Глава 17
Автоматизированное тестирование в браузере	Глава 17
Поисковая оптимизация	Глава 14
Изящный обход	Глава 13
Прогрессивное улучшение	Глава 10
Шаблоны клиенткой стороны	Глава 4
Мобильная разработка	Глава 10
Атаки межсайтовыми сценариями (XSS)	Глава 9
Подделка межсайтовых запросов (CSRF)	Глава 9

Тема	Глава (главы)
Атаки внедрением SQL-кода	Глава 9
Веб-службы и REST	Глава 7
Шаблон Repository	Глава 8
Непрерывная интеграция	Глава 18
Непрерывное развертывание	Глава 18
Развертывание в облаке/ферме	Глава 19
Кеширование серверной стороны	Глава 12
Кеширование клиентской стороны	Глава 12
Инструменты, инфраструктуры и технологии	Инструменты, инфраструктуры и технологии
jQuery	Глава 4
Клиентская проверка достоверности	Глава 3
Аутентификация/авторизация	Глава 9
Пакетирование и минимизация	Глава 13
ASP.NET Web API	Глава 7
Entity Framework	Главы 3, 8
jQuery Mobile	Глава 10
Веб-сокеты	Глава 11
SignalR	Глава 11
Windows Azure	Глава 19
Локальное хранилище браузера	Глава 12

Предметный указатель

A

AJAX (Asynchronous JavaScript and XML), 86; 117; 137; 190
API-интерфейс ApplicationCache, 252
ASP (Active Server Pages), 20
ASP.NET MVC
установка, 25

C

CDN (Content Delivery Network), 260
CI (Continuous Integration), 362
COM (Component Object Model), 295
CORS (Cross-Origin Resource Sharing), 137; 141; 142
CRUD (Create; Read; Update; Delete), 146; 158
CSRF (Cross-Site Request Forgery), 197
CSS (CSS Media Queries), 213

D

DI (Dependency Injection), 111
DIP (Dependency Inversion Principle), 108
DOM (Document Object Model), 78
DRY (Don't Repeat Yourself), 115
DTO (Data Transfer Object), 137

E

EBuy, 166; 372
бизнес-модель приложения Ebuy, 166
Entity Framework (EF), 160
Entity Tag (ETag), 271

G

Glimpse, 289

I

IIS (Internet Information Server), 23; 181; 372
создание и конфигурирование веб-сайта
IIS, 373
IoC (Inversion of Control), 108
ISP (Interface Segregation Principle), 106

J

JavaScript, 78; 123
JIT (Just-In-Time), 369
jQuery Mobile, 216
сдвиг парадигмы в jQuery Mobile, 216
jQuery Mobile Framework, 201
JSONP (JSON with Padding), 137

L

LINQ (Language Integrated Query), 170
LSP (Liskov Substitution Principle), 105

M

MVC (Model-View-Controller), 21; 94

N

NuGet, 391; 409

O

OCP (Open/Closed Principle), 104
OData (Open Data Protocol), 149
ORM (Object Relational Mapping), 69; 159

P

PI (Persistence Ignorance), 156
POCO (Plain Old CLR Object), 70; 156
PRG (Post-Redirect-Get), 416

R

RAD (Rapid Application Development), 60
Razor, 39; 41; 300
REST (Representational State Transfer), 137

S

SEO (Search Engine Optimization), 281
SignalR, 233
SRP (Single Responsibility Principle), 103
SOLID, 103

T

TDD (Test-driven development), 336

U

UAT (User Acceptance Testing), 328

W

WCF (Windows Communication Foundation), 103
Web Forms, 58

A

Абстракция, 409
Авторизация, 179
пользователей, 191
Архитектура
модель-представление-контроллер
(MVC), 21

Атака

- внедрением SQL-кода, 192
- межсайтовыми сценариями, 196
- Аутентификация, 53; 179
 - пользователей, 186
 - с помощью форм, 184

Б

- База данных
 - параллелизм базы данных, 162
- Безопасность, 177
 - управление безопасностью пользователей, 191
- Библиотека
 - Entity Framework (EF), 69
 - jQuery, 91
- Бизнес-модель приложения Ebuy, 166

В

- Веб-сокет, 228
- Визуализация
 - адаптивная, 210
 - частичная, 117
- Внедрение зависимостей (dependency injection), 108; 111

Д

- Данные
 - навигация по данным, 160
 - отправка данных на сервер, 132
 - сохранение данных в базе, 69
 - шаблоны доступа к данным, 156
- Действие, 33
- Диспетчер пакетов NuGet, 28; 409
- Доставки контента (CDN), 260

З

- Запрос
 - HTTP, 258
 - медиа-запросы CSS, 213
- Защита кеша, 276

И

- Имя представления
 - явное, 412
- Инверсия управления (IoC), 108
- Интеграция
 - непрерывная (CI), 362
- Интерфейс ApplicationCache, 252
- Иключение
 - обработка исключений, 150
- Исполняющая среда SignalR, 233

К

- Кеш
 - браузера, 251
 - вывода, 240
 - защита кеша, 263; 276
 - управление кешем, 389
- Кеширование, 235
 - “бублика”, 243
 - вывода, 419
 - “дырки от бублика”, 245
 - клиентской стороны, 236; 251
 - настройка клиентского кеширования в IIS, 263
 - настройка клиентского кеширования посредством ASP.NET MVC, 263
 - на уровне запросов, 236
 - на уровне пользователей, 237
 - на уровне приложений, 238
 - распределенное, 246
 - серверной стороны, 235
- Класс
 - AccountController, 54
 - AsyncController, 420
 - Auction, 167
 - AuctionsController, 111
 - BufferedMediaTypeFormatter, 153
 - ErrorLoggerManager, 104
 - HtmlHelper, 299
 - HttpResponseException, 150
 - JsonModelBinder, 134
 - MediaTypeFormatter, 153
 - RouteGenerator, 291
- Код
 - блок кода, 40
 - дублированный, 337
 - покрытие кода, 349
 - разработка поддающегося тестированию кода, 351
 - фрагменты кода (code nuggets), 40
- Компоновки, 41
- Контроллер, 23; 33; 47
 - AccountController, 185
 - асинхронный, 23; 222; 420
 - создание, 223
 - действие контроллера, 33
 - добавление контроллера Web API, 144
 - шаблоны контроллера, 48
- Конфигурирование IIS 7, 182
- Концентратор, 231

Л

Локатор служб (service locator), 108

М

“Магические строки”, 410

Маршрут, 47

конфигурирование маршрутов, 31

ограничения маршрутов, 287

универсальный, 286

Маршрутизация, 30; 59

конвойер маршрутизации, 293

на основе атрибутов, 289

приоритет маршрутизации, 284

расширенная, 279; 293

Мастер-страница, 64

Метод

вспомогательный, 63

расширяющий, 63

Механизм Razor, 41

Модель, 22; 46

предметной области, 168

Модули HTTP, 57

Мониторинг работоспособности ASP.NET, 318

Н

Наследование, 159

Нотация объектов JavaScript (JSON), 123

О

Обработка исключений, 150

Обработка ошибок в ASP.NET MVC, 312

Обработчики HTTP, 57

Обработчик ошибок

глобальный, 314

Обfuscация, 269

Объектно-реляционный отобразитель
(ORM), 159

Объекты CLR, 156

Оператор

if, 411

if-then-else, 415

Операции CRUD, 146

Очистка, 240

Ошибка

регистрация ошибок в журнале, 316

фильтры ошибок, 318

П

Пакет, 275

сборки, 398

Пакетирование, 275

Параллелизм

оптимистический, 163

пессимистический, 162

Перенаправление, 269

Платформа веб-разработки от Microsoft

Active Server Pages (ASP), 20

ASP.NET MVC, 21

ASP.NET Web Forms, 21

Поисковая оптимизация (SEO), 281

Представление, 22; 38; 50

частичное, 43

Приложение

Ebuy, 372

защита приложения, 179

построение защищенных

веб-приложений, 177

разработка веб-приложений для

мобильных устройств, 200

создание нового интранет-приложения

ASP.NET MVC, 180

создание нового мобильного приложения

с нуля, 216

Принцип

Don’t Repeat Yourself (DRY), 115

инверсии зависимостей (DIP), 108

подстановки Барбары Лисков (LSP), 105

разделения интерфейса (ISP), 106

Проект

тестовый, 329

Проектирование

логическое, 97

полезные советы, 99

физическое, 100

полезные советы, 101

Пространство имен, 100

Протокол

OData, 149

Путь

нахождение пути (wayfinding), 279

Р

Развертывание, 368

в Windows Azure, 378

на сервере IIS, 372

непрерывное, 381

сборок ASP.NET MVC Framework, 373

Разделения ресурсов между источниками

(CORS), 141

Репозиторий

обобщенный, 157

Рефакторинг модульных тестов, 342

C

Сайт

- перевод сайта Web Forms на ASP.NET MVC, 385
- создание и конфигурирование веб-сайта IIS, 373
- создание нового веб-сайта Windows Azure, 379

Сборка

- пакеты сборок, 398

Селектор, 80

Сервер построения, 357

Сериализация типа, 154

Словарь ViewData, 410

События, отправляемые сервером, 228

Сокет, 228

Состояние

- управление состоянием, 57

Среда SignalR, 233

Средство Glimpse, 289

Сценарий

- MSBuild, 376

- ленивая загрузка сценариев, 266

- минимизация сценариев, 268

- построения, 355

T

Тег сущности (ETag), 271

Тестирование

- автоматизированное, 324

- контроллеров, 339

- логики

- доступа к данным, 340

- приложения в браузере, 347

- модели, 333

- пользовательское приемочное, 328

- представлений, 347

- приложения ASP.NET MVC, 333

- разработка поддающегося тестированию кода, 351

- разработка через тестирование, 336

- ручное, 323

Тест

- интеграционный, 327

модульный, 324

рефакторинг модульных тестов, 342

написание чистых автоматизированных тестов, 337

приемочный, 328

Технология

Cross-Origin Resource Sharing (CORS), 142

Тип

MIME, 152

сериализация типа, 154

У

Установка ASP.NET MVC, 25

Ф

Файл

Global.asax, 417

NuSpec, 392; 394

web.config, 387

Фильтр

действий, 37

ошибок, 318

Форма

построение, 67

Х

Хранилище

локальное, 254

Ш

Шаблон

Repository (Репозиторий), 157

внедрения зависимостей, 111

доступа к данным, 156

клиентской стороны, 126

контроллера, 48

проекта, 26

проектирования Post-Redirect-Get (PRG), 416

Э

Элементы управления

серверные, 63

Я

Язык

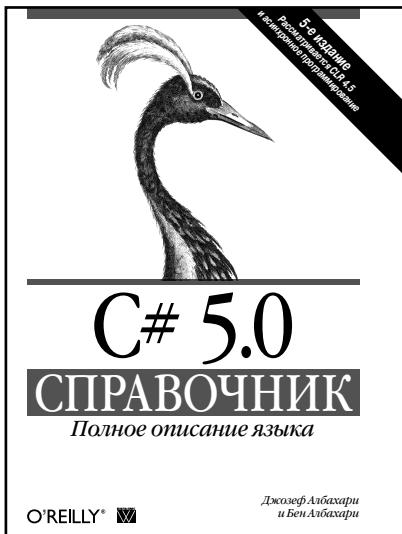
LINQ, 170

C# 5.0

СПРАВОЧНИК

ПОЛНОЕ ОПИСАНИЕ ЯЗЫКА

**Джозеф Албахари,
Бен Албахари**



www.williamspublishing.com

Данное руководство ставшее бестселлером, позволяет получить точные ответы практически на любые вопросы по C# 5.0 и .NET CLR. Уникально организованное по концепциям и сценариям использования, обновленное пятое издание книги предлагает реорганизованные разделы, посвященные параллелизму, многопоточности и параллельному программированию, а также включает подробные материалы по новому средству C# 5.0 — асинхронным функциям.

Проверенная более чем 20 экспертами, в числе которых Эрик Липперт, Стивен Тауб, Крис Барроуз и Джон Скит, эта книга содержит все, что необходимо для освоения C# 5.0. Она широко известна как исчерпывающий справочник по языку.

ISBN 978-5-8459-1819-2 в продаже

C# 5.0

КАРМАННЫЙ СПРАВОЧНИК

Джозеф Албахари
Бен Албахари



Книга является идеальным кратким справочником, позволяющим быстро найти исчерпывающую информацию по языку C# 5.0. В ней изложены все основные темы, касающиеся языка C# 5.0, как основы, так и более сложные темы, такие как перегрузка операторов, ограничения, ковариантность и контравариантность, итераторы, типы, допускающие нулевое значение, заимствование операторов, лямбда-выражения и замыкания. Кроме того, в книге изложена информация о языке LINQ, начиная с последовательностей, отложенного выполнения и стандартных операторов запроса и заканчивая полным справочником по выражениям запроса. Описаны динамическое связывание и новые асинхронные функции в языке C# 5.0, а также вопросы, касающиеся небезопасного кода и указатели, собственные атрибуты, директивы препроцессоров и документация XML.

www.williamspublishing.com

ISBN 978-5-8459-1820-8

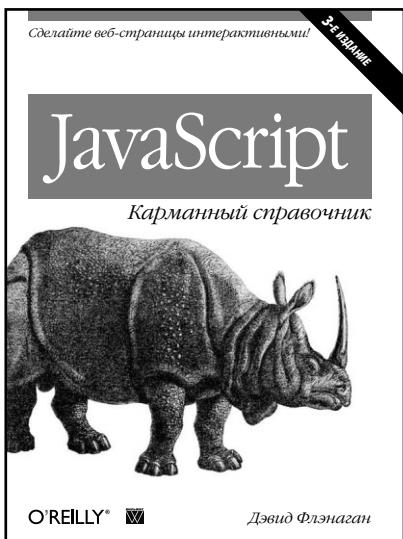
в продаже

JAVASCRIPT

КАРМАННЫЙ СПРАВОЧНИК

3-Е ИЗДАНИЕ

Дэвид Флэнаган



JavaScript – популярнейший язык программирования, который уже более 15 лет применяется для написания сценариев интерактивных веб-страниц. В книге представлены наиболее важные сведения о синтаксисе языка и показаны примеры его практического применения. Несмотря на малый объем карманного издания, в нем содержится все, что необходимо знать для разработки профессиональных веб-приложений. Главы 1–9 посвящены описанию синтаксиса последней версии языка (спецификация ECMAScript 5).

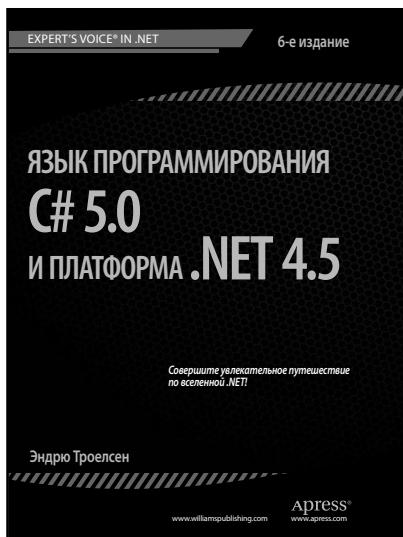
- Типы данных, значения и переменные
 - Инструкции, операторы и выражения
 - Объекты и массивы
 - Классы и функции
 - Регулярные выражения
- В главах 10–14 рассматриваются функциональные возможности языка наряду с моделью DOM и средствами поддержки HTML5.

www.williamspublishing.com

ISBN 978-5-8459-1830-7 в продаже

ЯЗЫК ПРОГРАММИРОВАНИЯ C# 5.0 И ПЛАТФОРМА .NET 4.5 6-Е ИЗДАНИЕ

Эндрю Троелсен



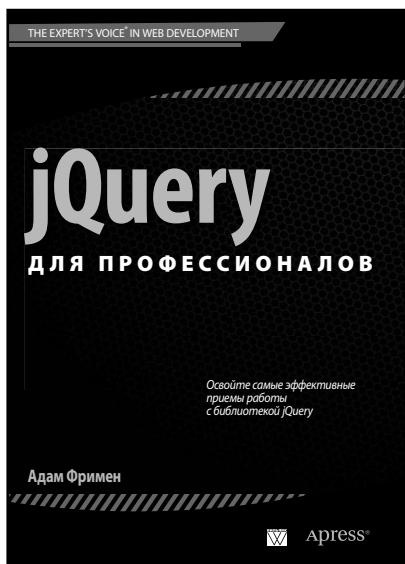
www.williamspublishing.com

Новое издание этой книги было полностью пересмотрено и переписано с учетом последних изменений в спецификации языка C# и дополнений платформы .NET Framework. Отдельные главы посвящены важным новым средствам, которые превращают .NET Framework 4.5 в самое передовое решение для корпоративных приложений. Помимо этого, рассмотрены все ключевые возможности языка C#, как старые, так и новые, что позволило обрасти популярность предыдущим изданиям этой книги (материал покрывает все темы, начиная с обобщений и кончая pLINQ). Основное назначение книги – служить исчерпывающим руководством по языку программирования C# и ключевым аспектам платформы .NET (сборкам, удаленному взаимодействию, Windows Forms, Web Forms, ADO.NET, веб-службам XML и т.д.).

ISBN 978-5-8459-1814-7 в продаже

jQUERY ДЛЯ ПРОФЕССИОНАЛОВ

Адам Фримен



В книге показано, как создавать профессиональные веб-приложения с меньшими усилиями и при меньшем размере кода. Вы изучите методы работы со встроенными и дистанционными данными, научитесь создавать функционально насыщенные интерфейсы для веб-приложений, а также познакомитесь с возможностями сенсорно-ориентированного фреймворка jQuery Mobile.

Основные темы книги:

- возможности и особенности библиотеки jQuery;
- применение базовых инструментов jQuery для улучшения HTML-документов, включения в них таблиц, форм и средств отображения данных;
- применение библиотеки jQuery UI для создания гибких и удобных в использовании веб-приложений;
- программирование различных элементов взаимодействия, как перетаскивание и вставка объектов, сортировка данных и сенсорная чувствительность;
- применение библиотеки jQuery Mobile при разработке сенсорно-ориентированных интерфейсов для мобильных устройств и планшетных компьютеров;
- расширение библиотеки jQuery путем создания собственных подключаемых модулей и виджетов.

www.williamspublishing.com

ISBN 978-5-8459-1799-7 В продаже

ASP.NET MVC 4: разработка реальных веб-приложений с помощью ASP.NET MVC

Научитесь работать с ASP.NET MVC 4 и узнайте, как писать современные серверные веб-приложения. Это руководство поможет понять, каким образом работает инфраструктура, и объяснит, как использовать различные средства для решения множества реальных сценариев разработки, с которыми чаще всего приходится сталкиваться. Кроме того, вы узнаете, как работать с HTML, JavaScript, Entity Framework и другими веб-технологиями.

Книга начинается с раскрытия ключевых концепций, таких как архитектурный шаблон “модель–представление–контроллер”, и продолжается рассмотрением более сложных тем. Авторы предлагают полезные советы и приемы для ASP.NET MVC 4, создавая на протяжении всей книги демонстрационный сайт онлайновых аукционов (“EBuy”).

- Узнайте о сходствах между ASP.NET MVC 4 и Web Forms
- Используйте Entity Framework для создания и обслуживания базы данных приложения
- Создавайте развитые веб-приложения, используя jQuery для разработки клиентской стороны
- Внедряйте технологии AJAX в свои веб-приложения
- Научитесь создавать и открывать доступ к службам ASP.NET Web API
- Обеспечьте развитый и полноценный пользовательский интерфейс для мобильных устройств
- Применяйте технологии обработки ошибок, автоматизированного тестирования и автоматизации построения
- Используйте различные варианты развертывания приложений ASP.NET MVC 4

Джесс Чедвик — независимый консультант по программному обеспечению, специализирующийся на веб-технологиях, который имеет звания ASPInsider и Microsoft MVP в ASP.NET.

Тодд Снайдер — ведущий консультант в компании Infragistics, специализирующийся на разработке многоуровневых и веб-приложений на платформе Microsoft.

Хришикеш Панда — разработчик настольных, веб-и мобильных приложений, имеющий более чем 14-летний опыт программирования.

“Эта книга представляет собой великолепный ресурс для разработчиков, желающих создавать веб-сайты с использованием ASP.NET MVC.”

Авторы делятся своим практическим опытом и помогают выбрать правильные технологии для работы, включая jQuery, Entity Framework и Web API”.

Марцин Добош,
старший разработчик, Microsoft

Категория: веб-разработка

Предмет рассмотрения: ASP.NET MVC

Уровень: Для пользователей средней и высокой квалификации

ISBN 978-5-8459-1841-3



13031
9 785845 918413



www.williamspublishing.com

O'REILLY®
oreilly.com