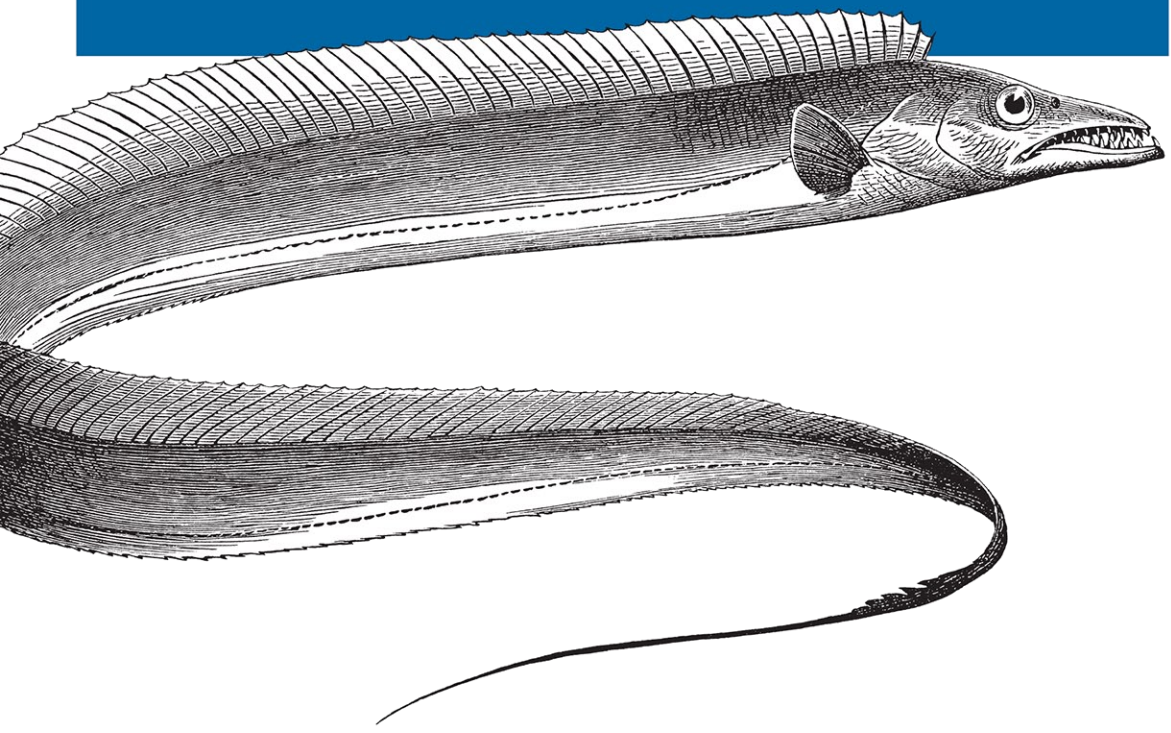
Разработка реальных веб-приложений с помощью ASP.NET MVC

ASP.NET MVC 4



Джесс Чедвик, Тодд Снайдер,
Хришикеш Панда

O'REILLY® ВИЛЬЯМС

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**, Тодд, **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

Научно-популярное издание

Джесс Чедвик, Тодд 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.)

Оглавление

ГЛАВА 1

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.

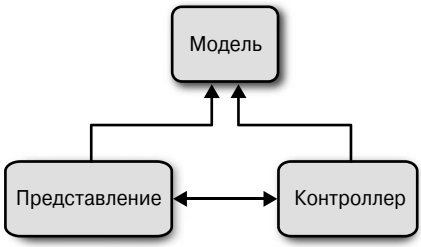


Рис. 1.1. *Архитектура MVC*

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 EBuy мы построим сайт онлайновых аукционов на ASP.NET MVC! С высокоуровневой 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, EBuy — 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 EBuy — от создания проекта и до развертывания, что позволит вам эффективно отслеживать весь процесс проектирования.

Полный исходный код приложения EBuy 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.

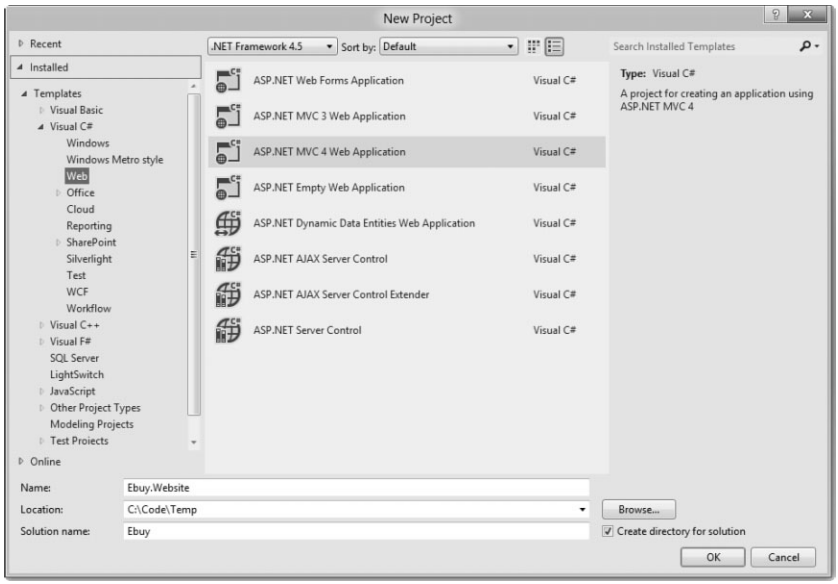


Рис. 1.2. *Создание проекта EBuy*

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 сгенерирует, 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 попутно. Этот шаблон также включает стандартную компоновку представления и генерирует файл `Global.asax`, включающий стандартный код конфигурации, который необходим большинству приложений ASP.NET MVC.

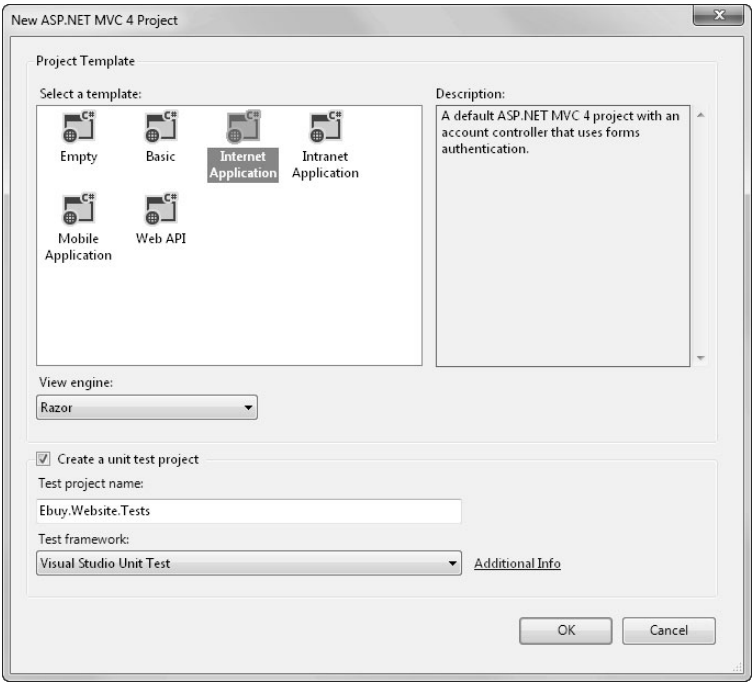


Рис. 1.3. *Настройка проекта EBuy*

- **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 (Интранет-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 EBuy 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 сгенерировать 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.

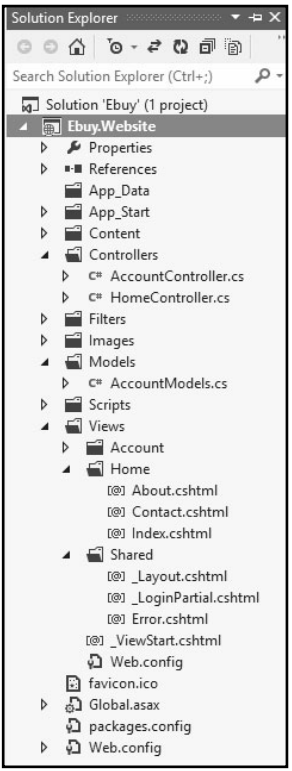


Рис. 1.4. *Структура папок проекта ASP.NET MVC*

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 тривиальной. 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 вебсайт 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.

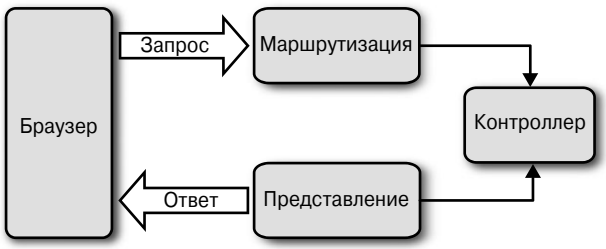


Рис. 1.5. *Жизненный цикл запроса ASP.NET MVC*

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 сопоставления 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).

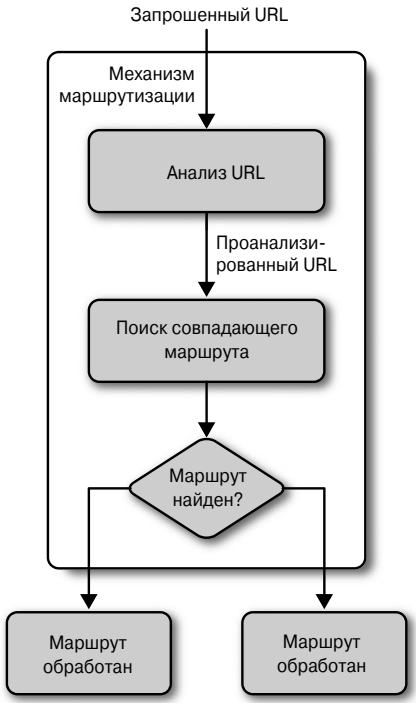


Рис. 1.6. *Маршрутизация ASP.NET*

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/rEceNt 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 системой 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 особенным; 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.
- `HttpNotFound()`. Возвращает экземпляр `HttpNotFoundResult`, 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 устранению *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 применимого 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`:

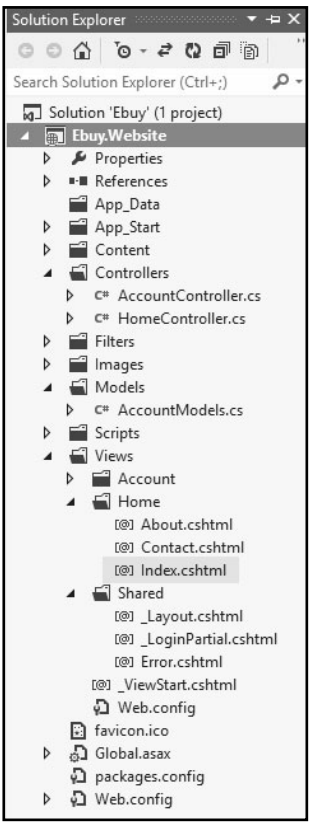


***Рис. 1.7.** Определение местоположения представления `Index`*

```
public ActionResult Index()
{
   ViewBag.Message = "Your app description page.";
   return View();
}
```

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`.

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.

Различение 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 использоваться 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 -->
@foreach(var bid in bids) {
   <!-- Переменная bid доступна только внутри цикла foreach -->
   <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
     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 *компоновок*. В этом случае единственное представление действует в качестве шаблона для всех других используемых представлений, определяя страничную 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>
```