# ГЛАВА 10

# Обзор проектов MVC

Прежде чем погрузиться в детали специфических средств MVC Framework, следует рассмотреть ряд дополнительных вопросов. В этой главе представлен обзор структуры и природы приложения ASP.NET MVC, включая структуру проекта по умолчанию и соглашения об именовании, которым необходимо следовать.

## Работа с проектами MVC в Visual Studio

Когда вы создаете новый проект MVC 3, среда Visual Studio предлагает выбрать одну из трех отправных точек: Empty (Пустой), Internet Application (Интернет-приложение) или Intranet Application (интранет-приложение), как показано на рис. 10.1.

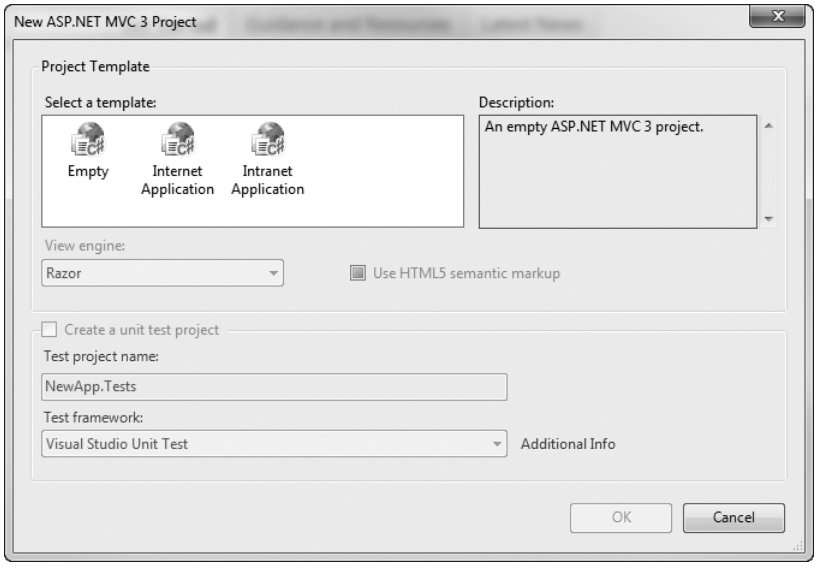


**Рис. 10.1.** Выбор начальной конфигурации проекта MVC 3

Шаблон проекта Empty использовался для приложения RSVP в главе 3 и при настройке приложения SportsStore в главе 7. Он создает относительно небольшой набор файлов и каталогов, а также предоставляет минимальную структуру, на основе которой будет строиться приложение.

**На заметку!** Диалоговое окно New ASP.NET MVC 3 Project (Новый проект ASP.NET MVC 3), показанное на рис. 10.1, содержит флажок Use HTML5 semantic markup (Использовать разметку с семантикой HTML5). В Microsoft начали процесс добавления поддержки HTML5 к Visual Studio. Однако в этой книге мы решили проигнорировать HTML5. Платформа MVC Framework не зависит от версии языка HTML, используемой в проекте. HTML5 представляет собой самостоятельную тему. Если вы заинтересованы в изучении HTML5, обратитесь к книге Адама Фримена под названием *The Definitive Guide to HTML5* (Apress).

Шаблоны проектов Internet Application и Intranet Application заполняют проект, чтобы предоставить более полную стартовую точку, используя различные механизмы аутентификации, которые подходят для Интернет- и интранет-приложений (системы аутентификации более подробно рассматриваются в главе 22).

**Совет.** В случае использования шаблона Internet Application или Intranet Application также можно создать проект модульного тестирования как часть решения Visual Studio, отметив флажок Create a unit test project (Создать проект модульного тестирования). Это еще одно удобство, поскольку проект тестирования можно создать и самостоятельно, как это делалось в главе 7.

Различия между этими тремя стартовыми опциями показаны на рис. 10.2, где настроен начальный проект для упомянутых трех шаблонов.

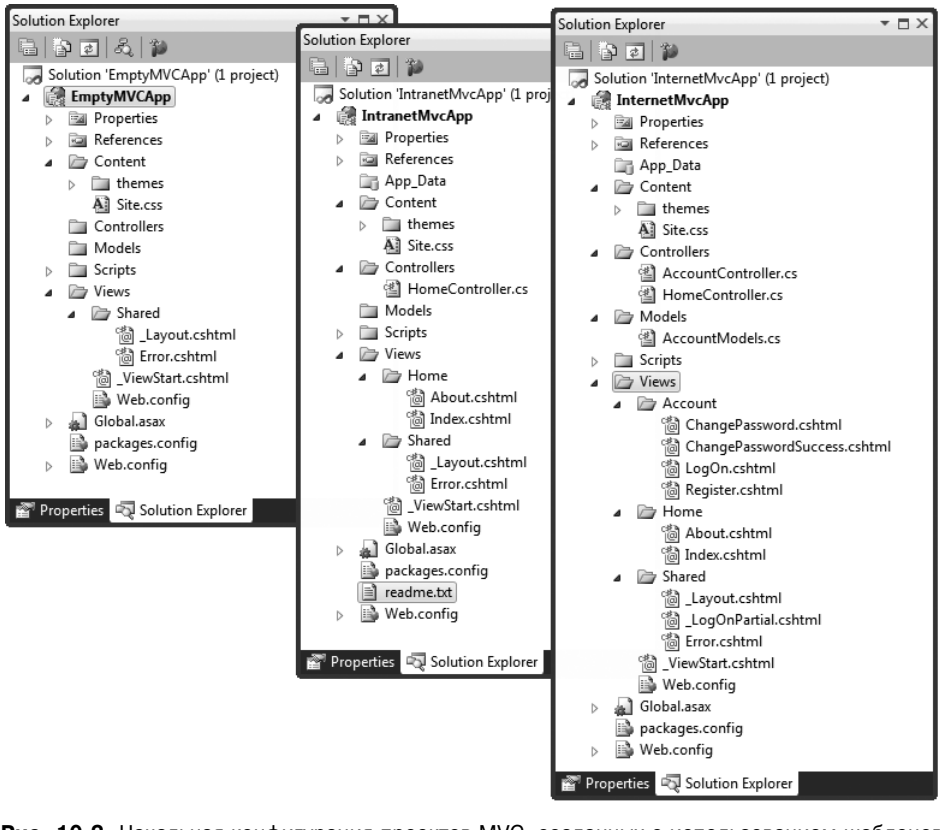


**Рис. 10.2.** Начальная конфигурация проектов MVC, созданных с использованием шаблонов Empty, Internet Application и Intranet Application

Шаблоны создают проекты, которые имеют общую структуру. Некоторые из элементов проекта играют особые роли, жестко закодированные в ASP.NET или MVC Framework. Другие имеют отношение к соглашениям об именовании. Все эти файлы и папки описаны в табл. 10.1.

**Таблица 10.1. Сводка по элементам проекта MVC 3**

| Папка или файл | Описание | Примечания |
|---|---|---|
| `/App_Data` | В этот каталог помещаются приватные данные, такие как XML-файлы или базы данных, если используется SQL Server Express, SQLite или какой-то другой репозиторий на основе файлов | Веб-сервер IIS не будет обслуживать содержимое этого каталога |
| `/bin` | Сюда помещается скомпилированная сборка MVC-приложения вместе со всеми ссылаемыми сборками, находящимися не в GAC | Веб-сервер IIS не будет обслуживать содержимое этого каталога. Чтобы увидеть каталог `bin` в окне Solution Explorer (Проводник решения), понадобится щелкнуть на кнопке Show All Files (Показать все файлы). Поскольку все файлы в этом каталоге являются двоичными файлами, сгенерированными в результате компиляции, обычно они не хранятся в системе управления исходным кодом |
| `/Content` | Сюда помещается статический контент, такой как CSS-файлы и изображения | Это является необязательным соглашением. Статический контент можно хранить в любом подходящем месте |
| `/Controllers` | Сюда помещаются классы контроллеров | Это является соглашением. Классы контроллеров могут размещаться где угодно, поскольку они компилируются в ту же самую сборку |
| `/Models` | Сюда помещаются классы модели представления и модели предметной области, хотя все кроме простейших приложений выигрывают от определения модели предметной области в отдельном проекте, как было продемонстрировано в приложении SportsStore | Это является соглашением. Классы моделей могут быть определены где угодно в текущем проекте или вообще в отдельном проекте |
| `/Scripts` | Этот каталог предназначен для хранения JavaScript-библиотек приложения. По умолчанию Visual Studio добавляет библиотеки для вспомогательных классов jQuery и Microsoft AJAX | Это является соглашением. Файлы сценариев могут находиться в любом месте, т.к. в действительности они представляют собой просто другой тип статического контента |

*Окончание табл. 10.1*

| Папка или файл | Описание | Примечания |
|---|---|---|
| /Views | В этом каталоге хранятся представления и частичные представления, обычно сгруппированные вместе в папках с именем контроллера, с которым они связаны | Файл /Views/Web.config предотвращает обслуживание веб-сервером IIS содержимого этих каталогов. Представления должны визуализироваться через методы действий |
| /Views/Shared | В этом каталоге хранятся компоновки и представления, не являющиеся специфичными для какого-либо контроллера | |
| /Views/Web.config | Это *не* конфигурационный файл для приложения. В нем содержится конфигурационная информация, которая обеспечивает обработку представлений с помощью ASP.NET и запрещает их обслуживание веб-сервером IIS | |
| /Global.asax | Это глобальный класс приложения ASP.NET. В его классе отделенного кода (Global.asax.cs) осуществляется регистрация конфигурации маршрутов, а также настройка любого кода, который должен выполняться при запуске или завершении приложения либо в случае возникновения необработанного исключения | В MVC-приложении файл Global.asax играет ту же самую роль, что и в приложении Web Forms |
| /Web.config | Конфигурационный файл для приложения. Его роль более подробно рассматривается далее в этой главе | В MVC-приложении файл Web.config играет ту же самую роль, что и в приложении Web Forms |

**На заметку!** Как будет показано в главе 23, приложение MVC развертывается копированием этой структуры папок на веб-сервер. Из соображений безопасности IIS не обслуживает файлы, полный путь которых содержит Web.config, bin, App_code, App_GlobalResources, App_LocalResources, App_WebReferences, App_Data или App_Browsers. Веб-сервер IIS также будет игнорировать запросы файлов с расширениями .asax, .ascx, .sitemap, .resx, .mdb, .mdf, .ldf, .csproj и некоторыми другими. Если вы решили реструктурировать свой проект, то должны удостовериться, что упомянутые имена и расширения файлов не используются в URL-адресах.

В табл. 10.2 описаны папки и файлы, которые в случае их существования имеют специальное значение в проекте MVC 3.

**Таблица 10.2. Сводка по дополнительным элементам проекта MVC 3**

| Папка или файл | Описание |
|---|---|
| /Areas | Области — это способ разбиения крупного приложения на небольшие порции. Работа областей рассматривается в главе 11 |
| /App_GlobalResources<br>/App_LocalResources | Эти папки содержат ресурсные файлы, используемые для локализации страниц Web Forms |
| /App_Browsers | Эта папка содержит XML-файлы .browser, которые описывают, как идентифицировать специфические веб-браузеры, а также их возможности (например, поддержку JavaScript) |
| /App_Themes | Эта папка содержит темы Web Forms (включая файлы .skin), которые влияют на визуализацию элементов управления Web Forms |

**На заметку!** За исключением `/Areas`, элементы в табл. 10.2 являются частью ядра платформы ASP.NET и не имеют отношения к приложениям MVC. За подробным описанием средств платформы ASP.NET обращайтесь к книге *Microsoft ASP.NET 4.0 с примерами на C# 2010 для профессионалов* (ИД "Вильямс", 2011 г.).

## Использование контроллеров Интернет- и интранет-приложений

На рис. 10.2 видно, что шаблоны Internet Application и Intranet Application добавляют некоторые контроллеры, представления, компоновки и модели представлений по умолчанию. При этом включается довольно много функциональности, особенно в проектах, созданных с использованием шаблона Intranet Application.

Класс `HomeController` (представленный в обоих шаблонах Internet Application и Intranet Application) может визуализировать страницы `Home` и `About`. Эти страницы генерируются с компоновкой по умолчанию, которая использует CSS-файл с темой на основе приглушенного синего цвета.

Шаблон Internet Application также включает класс `AccountController`, который позволяет посетителям регистрироваться и входить в систему. Он применяет аутентификацию с помощью форм для отслеживания входа в систему и ключевое средство членства ASP.NET (рассматриваемое в главе 22) для ведения списка зарегистрированных пользователей. Средство членства попробует создать на лету файловую базу данных SQL Server Express в папке `/App_Data` при первой попытке кого-либо зарегистрироваться или войти в систему. Если СУБД SQL Server Express не установлена или не функционирует, по прошествии длительной паузы создание базы данных завершится неудачей. Контроллер `AccountController` также имеет действия и представления, которые позволяют зарегистрированным пользователям менять свои пароли. (В шаблоне Intranet Application контроллер `AccountController` отсутствует, поскольку предполагается, что учетные записи и пароли управляются инфраструктурой домена Windows/Active Directory.)

Контроллеры и представления по умолчанию могут быть полезны для быстрого запуска проекта, но мы предпочитаем использовать шаблон Empty, чтобы приложение содержало только те элементы, которые необходимы.

## Соглашения в MVC

В проекте MVC применяются два вида соглашений. Соглашения первого вида — это в действительности лишь предположения о том, как должна выглядеть структура проекта. Например, общепринято размещать файлы JavaScript в папке `Scripts`. Здесь их рассчитывают обнаружить другие разработчики, использующие MVC, и сюда Visual Studio помещает начальные файлы JavaScript для нового проекта MVC. Однако вы вольны переименовать папку `Scripts` или вообще удалить ее, разместив файлы сценариев в любом другом месте по своему выбору. Это не помешает MVC Framework запустить ваше приложение.

Соглашения второго вида проистекают из принципа *соглашения по конфигурации* (convention over configuration), который был одним из главных аспектов, обеспечивших популярность платформе Ruby on Rails. Соглашение по конфигурации означает, что вы не должны явно конфигурировать ассоциации между контроллерами и их представлениями. Нужно просто следовать определенному соглашению об именовании — и все будет работать. При соглашении такого рода существенно снижается гибкость в изменении структуры проекта. В последующих разделах объясняются соглашения, которые используются вместо конфигурации.

**Совет.** Как будет показано в главе 15, все соглашения могут быть изменены с помощью специального механизма визуализации.

### Следование соглашениям для классов контроллеров

Классы контроллеров должны иметь имена, заканчивающиеся на `Controller`, например, `ProductController`, `AdminController` и `HomeController`.

При ссылке на контроллер из маршрута MVC или вспомогательного метода HTML указывается первая часть имени (такая как `Product`), а класс `DefaultControllerFactory` автоматически добавит `Controller` к имени и начнет поиск класса контроллера. Это поведение можно изменить, создав собственную реализацию интерфейса `IControllerFactory`, который описан в главе 14.

### Следование соглашениям для представлений

Представления и частичные представления должны располагаться в папке `/Views/`*`Имя_контроллера`*. Например, представление, ассоциированное с классом `ProductController`, должно находиться в папке `/Views/Product`.

**На заметку!** Обратите внимание, что часть `Controller` класса в имени папки под `Views` не указывается, т.е. используется папка `/Views/Product`, *а не* `/Views/ProductController`. Поначалу это может показаться нелогичным, но очень скоро это войдет в привычку.

Платформа MVC Framework ожидает, что представление по умолчанию для метода действия должно получить имя этого метода. Например, представление, ассоциированное с методом действия `List`, должно называться `List.cshtml` (или `List.aspx`, если используется унаследованный механизм визуализации ASPX). Таким образом, ожидается, что для метода действия `List` в классе `ProductController` представлением по умолчанию будет `/Views/Product/List.cshtml`.

Представление по умолчанию используется при возврате результата вызова метода `View` в методе действия, примерно так:

```
return View();
```

Можно указать имя другого представления, как показано ниже:

```
return View("MyOtherView");
```

Обратите внимание, что мы не включаем в представление расширение имени файла или путь. MVC Framework попытается найти представление, используя расширения имени файла для установленных механизмов визуализации (Razor и механизм ASPX по умолчанию).

При поиске представления MVC Framework просматривает папку, имеющую имя контроллера, и затем папку `/Views/Shared` . Это значит, что представления, используемые более чем одним контроллером, можно поместить в папку `/Views/Shared` и платформа найдет их самостоятельно.

### Следование соглашениям для компоновок

Соглашение об именовании для компоновок предусматривает добавление к имени файла символа подчеркивания (как объяснялось в главе 9, это берет начало из платформы WebMatrix, которая также использует Razor), и файлы компоновки помещаются в папку `/Views/Shared`. Среда Visual Studio создает компоновку по имени `_Layout.cshtml` как часть начального шаблона проекта. Эта компоновка применяется ко всем представлениям по умолчанию, через файл `/Views/_ViewStart.cshtml`, который обсуждался в главе 5.

Если вы не хотите применения компоновки по умолчанию к представлениям, можете изменить настройки в `_ViewStart.cshtml` (или вообще удалить этот файл), указав другую компоновку в представлении, например:

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

Можно также отключить компоновку для заданного представления:

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

# Отладка приложений MVC

Отладка приложения ASP.NET MVC может быть организована тем же способом, что и отладка приложения ASP.NET Web Forms. Отладчик Visual Studio является мощным и гибким инструментом, обладающим множеством средств. В этой книге мы рассмотрим его лишь поверхностно. Мы покажем, как настраивать отладчик, создавать точки останова, а также запускать отладчик для приложения и модульных тестов.

## Создание проекта

Для демонстрации применения отладчика мы должны создать новый проект MVC 3 с использованием шаблона Internet Application. Это предоставит в пользование начальный контроллер и представления. Мы назовем проект `DebuggingDemo` и отметим флажок Create a unit test project (Создать проект модульного тестирования), как показано на рис. 10.3.

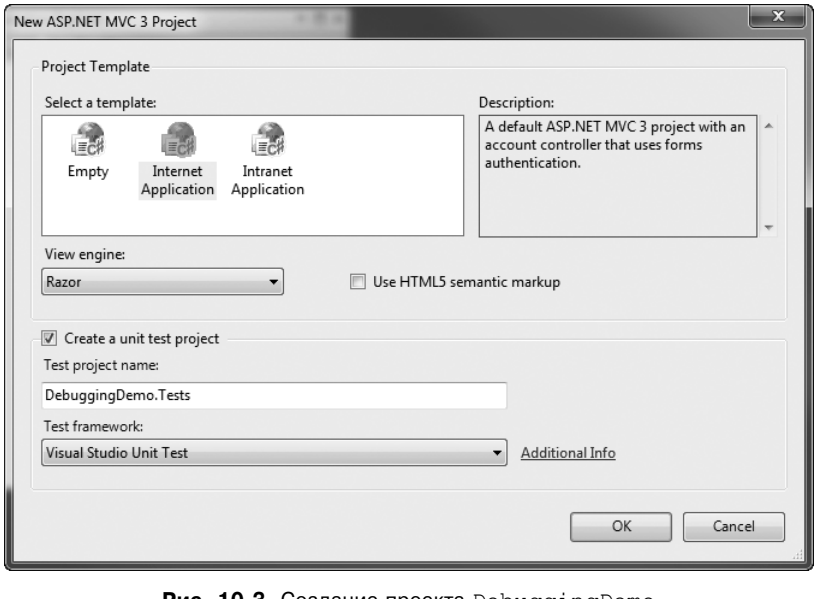


**Рис. 10.3.** Создание проекта `DebuggingDemo`

## Запуск отладчика Visual Studio

Перед тем как можно будет отлаживать приложение MVC, потребуется проверить используемую конфигурацию в Visual Studio. Классы C# (такие как контроллеры и сущности модели предметной области) должны быть скомпилированы в режиме отладки. Меню для установки этой опции показано на рис. 10.4; вариант Debug (Отладка) принят по умолчанию.

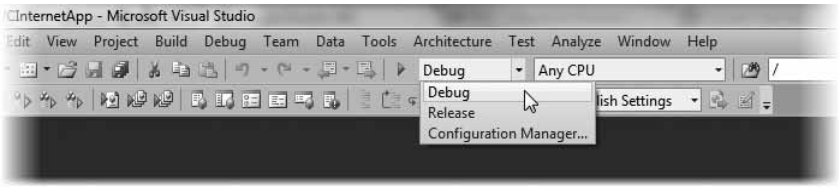


**Рис. 10.4.** Выбор конфигурации для отладки

Для запуска отладчика проще всего нажать клавишу <F5>. В качестве альтернативы можно выбрать пункт Start Debugging (Запустить отладку) в меню Debug (Отладка) среды Visual Studio. При запуске отладчика для приложения в первый раз может отобразиться диалоговое окно, показанное на рис. 10.5.

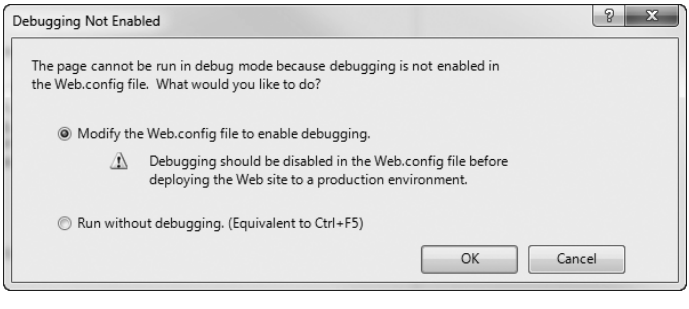


**Рис. 10.5.** Включение отладки в приложении MVC

Если выбран переключатель Modify the Web.config file to enable debussing (Модифицировать файл `Web.config`, чтобы включить отладку), раздел `compilation` в этом файле будет обновлен так, что значением атрибута `debug` станет `true`, как показано в листинге 10.1. При желании это значение можно изменить вручную.

**Листинг 10.1. Включение отладки в файле `Web.config`**

```
<configuration>
...
<system.web>
  <compilation debug="true" targetFramework="4.0">
  ...
  </compilation>
</system.web>
</configuration>
```

**Внимание!** Не развертывайте приложение на производственном сервере, предварительно не отключив в нем отладку. В главе 23 будут объяснены причины и показаны способы автоматизации этого изменения в виде части процесса развертывания.

В этот момент приложение запустится и отобразится в новом окне браузера. Отладчик подключится к приложению, но вы не заметите никакой разницы до тех пор, пока не произойдет останов при отладке (что это значит — объясняется в следующем разделе). Чтобы остановить отладчик, выберите пункт Stop Debugging (Остановить отладку) в меню Debug среды Visual Studio.

## Останов отладчика Visual Studio

Приложение, выполняющееся с присоединенным отладчиком, будет вести себя нормально до тех пор, пока не произойдет *останов*, при котором оно перестает выполняться, а управление возвращается отладчику. В этой точке вы можете проверять и управлять состоянием приложения. Остановы происходят по двум основным причинам: достигнута точка останова и случилось необработанное исключение. В следующих разделах будут приведены примеры обеих ситуаций.

**Совет.** Отладчик можно остановить вручную, выбрав пункт Break All (Остановить все) в меню Debug среды Visual Studio во время его выполнения.

### *Использование точек останова*

*Точка останова* (breakpoint) — это инструкция, которая сообщает отладчику о необходимости остановить выполнение приложения и передать управление программисту. После этого вы можете проинспектировать состояние приложения и посмотреть, что произошло. Для демонстрации точки останова мы добавим несколько операторов в метод `Index` класса `HomeController` , как показано в листинге 10.2.

**Листинг 10.2. Дополнительные операторы в классе `HomeController`**

```
using System.Web.Mvc;

namespace DebuggingDemo.Controllers {

    public class HomeController : Controller {

        public ActionResult Index() {
            int firstVal = 10;
            int secondVal = 5;
            int result = firstVal / secondVal;

            ViewBag.Message = "Welcome to ASP.NET MVC!";

            return View(result);
        }
        public ActionResult About() {
            return View();
        }
    }
}
```

Эти операторы не делают ничего особо интересного. Они были включены только для целей демонстрации некоторых средств отладчика. Для этой демонстрации мы хотим добавить точку останова к оператору, который устанавливает значение свойства `ViewBag.Message`. Чтобы создать точку останова, щелкните правой кнопкой мыши на операторе кода и выберите в контекстном меню пункт Breakpoint⇨Insert Breakpoint (Точка останова⇨Вставить точку останова). Слева от оператора появится красная точка, как показано на рис. 10.6.

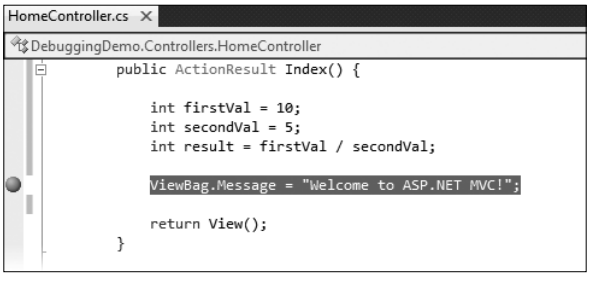


**Рис. 10.6.** Добавление точки останова

После запуска приложения в отладчике (выбором пункта Start Debugging в меню Debug) приложение будет выполняться вплоть до достижения оператора с этой точкой останова, а затем управление вернется вам (рис. 10.7).

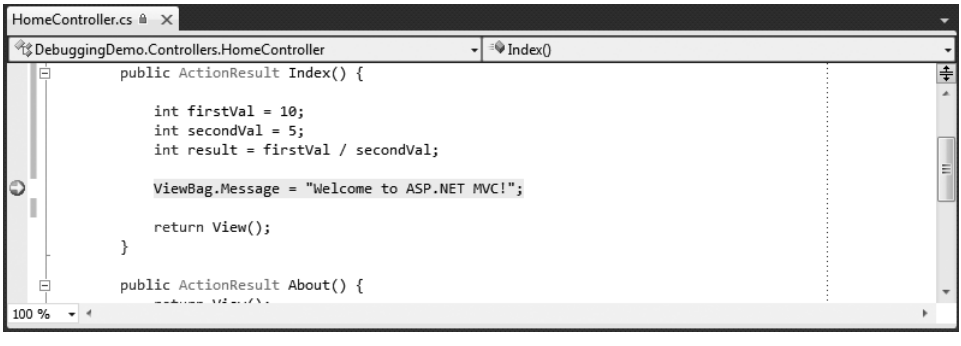


**Рис. 10.7.** Попадание в точку останова

---

**На заметку!** Точка останова активизируется, только когда выполняется ассоциированный с ней оператор. Рассматриваемый пример точки останова достигается сразу после запуска приложения, поскольку она находится внутри метода действия, который вызывается при получении запроса к URL по умолчанию. Если поместить точку останова внутрь другого метода действия, понадобится с помощью браузера запросить URL, ассоциированный с данным методом. Это может означать работу с приложением так, как это делает пользователь, или навигацию непосредственно к URL в окне браузера.

---

Точки останова можно размещать в месте, где подозревается наличие проблем в приложении. Как только точка останова достигнута, можно просмотреть стек вызовов, который привел к текущему методу, значения полей и переменных и многое другое. На рис. 10.8 показаны два способа просмотра значений переменных, добавленных к методу `Index`: использование окна Locals (Локальные) и наведение курсора мыши на имя переменной в окне кода.

Вы можете также управлять выполнением приложения. Перетаскивая стрелку желтого цвета, появляющуюся возле точки останова, можно указать оператор, который будет выполняться следующим. Вдобавок можно использовать пункты Step Into (Выполнить с заходом), Step Over (Выполнить без захода) и Step Out (Выполнить текущий метод) меню Debug.

Чтобы удалить точку останова, щелкните правой кнопкой мыши на операторе кода и выберите в контекстном меню пункт Breakpoint⇨Delete Breakpoint (Точка останова⇨Удалить точку останова). Чтобы удалить все точки останова, выбурите пункт Delete All Breakpoints (Удалить все точки останова) в меню Debug.

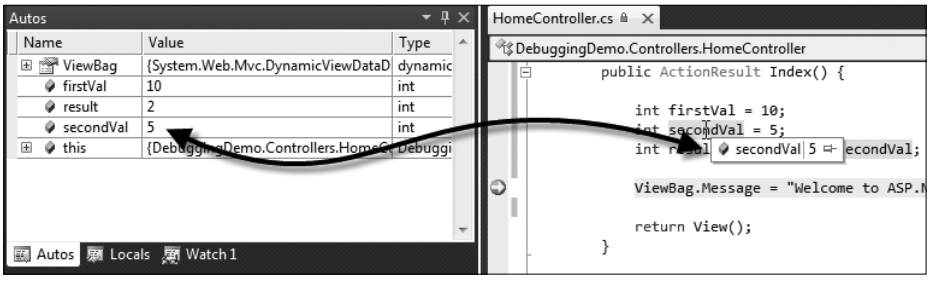


**Рис. 10.8.** Просмотр значений переменных в отладчике

**Совет.** Точки останова можно добавлять в представления для отладки представлений Razor. Например, они очень полезны при анализе значений свойств, относящихся к модели представления. Добавление точки останова в представление осуществляется точно так же, как в файл кода: щелкните правой кнопкой мыши на интересующем операторе Razor и выберите в контекстном меню пункт Breakpoint⇨Insert Breakpoint (Точка останова⇨Вставить точку останова).

### *Остановы из-за исключений*

Необработанные исключения — это явление процесса разработки. Одна из причин проведения модульного и интеграционного тестирования в проектах — сведение к минимуму вероятности возникновения таких исключений в производственных версиях. Отладчик Visual Studio останавливается автоматически, когда встречает необработанное исключение.

**На заметку!** К останову отладчика приводят только *необработанные* исключения. Исключение становится *обработанным*, если оно было перехвачено и обработано в блоке `try...catch`. Обработанные исключения представляют собой полезный инструмент в программировании. Они используются для моделирования сценария, при котором метод не может завершить свою задачу и должен уведомить об этом вызывающий код. Необработанные исключения нежелательны, т.к. они представляют неожиданное условие, которое не было скомпенсировано (и поскольку они переносят пользователя на страницу ошибки).

Для демонстрации останова по исключению мы внесли небольшое изменение в метод действия `Index`, как показано в листинге 10.3.

**Листинг 10.3. Добавление оператора, который вызовет исключение**

```
using System.Web.Mvc;
namespace DebuggingDemo.Controllers {
   public class HomeController : Controller {
      public ActionResult Index() {
         ViewBag.Message = "Welcome to ASP.NET MVC!";
         int firstVal = 10;
         int secondVal = 0;
         int result = firstVal / secondVal;
         return View(result);
      }
      public ActionResult About() {
         return View();
      }
   }
}
```

Мы изменили значение переменной `secondVal` на `0`, что приведет к исключению в операторе, где `firstVal` делится на `secondVal`. После запуска отладчика приложение будет выполняться вплоть до генерации исключения. Затем отобразится вспомогательное окно исключения, показанное на рис. 10.9.

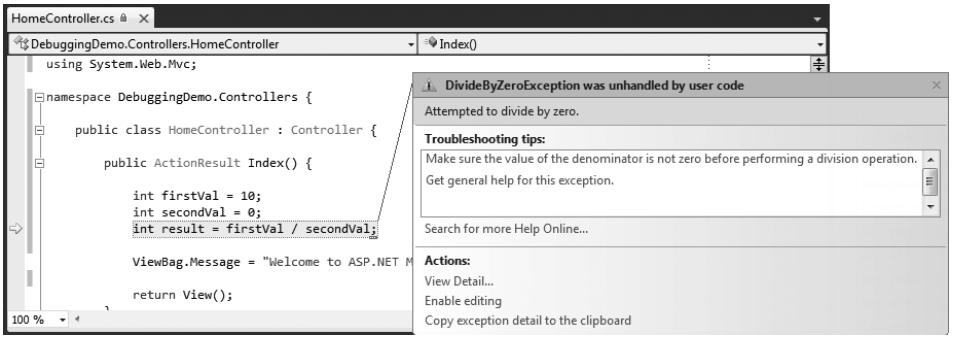


**Рис. 10.9.** Вспомогательное окно исключения

Вспомогательное окно исключения предоставляет детали, связанные с исключением. Когда отладчик останавливается по исключению, вы можете проинспектировать состояние приложения и потока управления, как это обычно делается при достижении точки останова.

## Использование средства Edit and Continue

Одно из наиболее интересных средств отладки Visual Studio называется *Edit and Continue* (Отредактировать и продолжить). Когда отладчик останавливается, вы можете отредактировать код и продолжить отладку. Среда Visual Studio перекомпилирует приложение и воссоздаст состояние приложения на момент останова отладки.

### *Включение средства Edit and Continue*

Средство Edit and Continue должно быть включено в двух местах.

- В разделе Edit and Continue (Отредактировать и продолжить) параметров Debugging (Отладка) диалогового окна Options (Параметры) (выберите пункт Options в меню Tools (Сервис) среды Visual Studio) удостоверьтесь, что флажок Enable Edit and Continue (Включить средство Edit and Continue) отмечен, как показано на рис. 10.10.

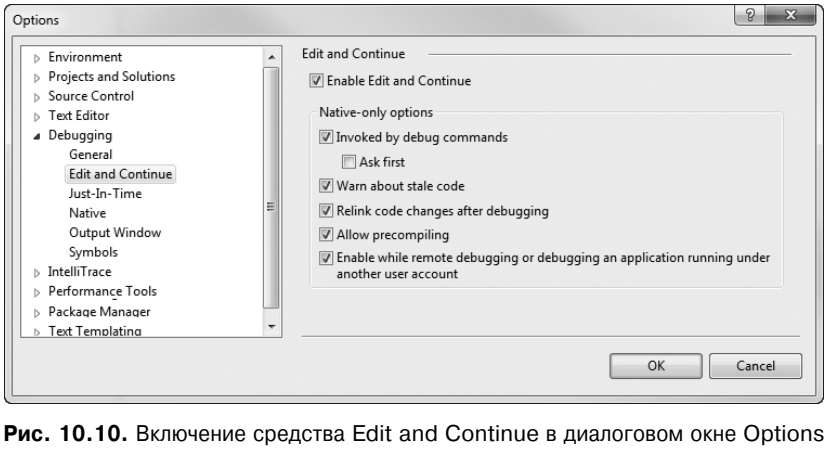


**Рис. 10.10.** Включение средства Edit and Continue в диалоговом окне Options

- В окне свойств проекта (выберите пункт <projectname> Properties (Свойства <имя_проекта>) в меню Project (Проект) среды Visual Studio) щелкните на разделе Web (Веб) и удостоверьтесь, что флажок Enable Edit and Continue отмечен, как показано на рис. 10.11.

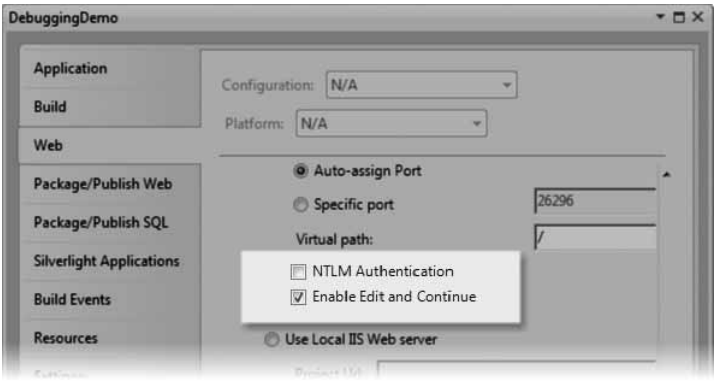


**Рис. 10.11.** Включение средства Edit and Continue в окне свойств проекта

### *Внесение модификаций в проект*

Средство Edit and Continue несколько придирчиво. Существуют условия, при которых оно работать не будет. Одно из таких условий присутствует в методе действия `Index` класса `HomeController`: использование динамических объектов. В частности, здесь с помощью `ViewBag` устанавливается сообщение для отображения представлением. Мы заменим это обращением к `ViewData`, как показано (и выделено полужирным) в листинге 10.4.

**Листинг 10.4. Удаление вызова `ViewBag` из метода `Index`**

```
public ActionResult Index() {

    int firstVal = 10;
    int secondVal = 0;
    int result = firstVal / secondVal;

    ViewData["Message"] = "Welcome to ASP.NET MVC!";

    return View(result);
}
```

В представление `Index.cshtml` также понадобится внести соответствующее изменение (см. листинг 10.5).

**Листинг 10.5. Удаление вызова `ViewBag` из представления**

```
@model int
@{
    ViewBag.Title = "Home Page";
}
<h2>@ViewData["Message"]</h2>
<p>
    The calculation result value is: @Model
</p>
```

Мы также воспользовались возможностью сделать представление строго типизированным и отобразили результат расчета, выполненного в методе `Index`.

### Демонстрация средства Edit and Continue

Теперь все готово для демонстрации средства Edit and Continue. Начните с выбора пункта Start Debugging в меню Debug. Приложение запустится с присоединенным отладчиком, и будет выполняться вплоть до достижения строки в методе `Index`, где производится простое вычисление. Значением второго параметра является `0`, поэтому генерируется исключение. В этой точке отладчик останавливает выполнение и отображается вспомогательное окно исключения (см. рис. 10.9).

В этом окне щелкните на ссылке Enable editing (Включить редактирование) и затем измените оператор, реализующий вычисление, как показано в листинге 10.6.

**Листинг 10.6. Редактирование актуального кода**

```
public ActionResult Index() {

    int firstVal = 10;
    int secondVal = 0;
    int result = firstVal / 5;

    ViewData["Message"] = "Welcome to ASP.NET MVC!";

    return View(result);
}
```

Выберите пункт Continue (Продолжить) в меню Debug. Приложение продолжит выполнение, а браузер отобразит визуализированную страницу, как показано на рис. 10.12.

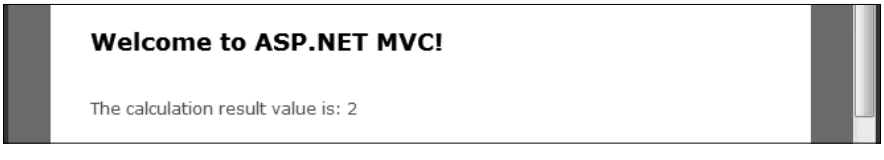


**Рис. 10.12.** Эффект от использования средства Edit and Continue

Потратьте некоторое время на обдумывание того, что здесь произошло. Мы запустили приложение с ошибкой внутри — попыткой деления на ноль. Отладчик обнаружил исключение и остановил выполнение программы. Мы отредактировали код, чтобы исправить ошибку, заменив ссылку на переменную литеральным значением `5`. Затем мы сообщили отладчику о необходимости продолжить выполнение.

В этот момент Visual Studio перекомпилирует приложение, чтобы включить изменение в процесс сборки, перезапустит выполнение, воссоздаст состояние, которое предшествовало исключению, и затем продолжит все как обычно. Браузер получит визуализированный результат, отражающий внесенные коррективы.

Без средства Edit and Continue пришлось бы остановить приложение, внести необходимые изменения, скомпилировать приложение и перезапустить отладчик. Затем понадобилось бы повторить в браузере все шаги, которые были проделаны до момента останова отладчика. Избежание этого последнего действия может быть очень важным. Сложные ошибки могут требовать множества шагов, связанных с воссозданием приложения, и возможность протестировать потенциальные исправления без необходимости постоянного повторения этих шагов экономит время и усилия программиста.

### Отладка модульных тестов

Отладчик Visual Studio можно использовать для отладки модульных тестов. Для этого понадобится выбрать одну из опций в меню Test⇨Debug (Тест⇨Отладить). Это подобно запуску модульных тестов, но с присоединенным отладчиком. Хоть такая возможность может показаться странной, но существуют две ситуации, в которых данное средство полезно.

- Когда вы получаете неожиданное или несовместимое поведение модульного теста. Если подобное случается, можно воспользоваться точками останова, чтобы прервать выполнение модульного теста и посмотреть, что происходит.
- Когда необходимо узнать больше о состоянии теста в случае его сбоя. С ситуацией подобного рода мы сталкиваемся чаще всего. Методы `Assert`, которые применяются для проверки результата модульного теста, генерируют исключение, если специфическое условие не удовлетворяется. Поскольку отладчик останавливается, когда сталкивается с необработанным исключением, мы можем включить отладку модульного теста, подождать, пока он не даст сбой, и затем посмотреть, что произошло вплоть до момента сбоя утверждения.

# Внедрение зависимости на уровне проекта

В последующих главах вы увидите множество различных способов, предлагаемых MVC Framework для расширения и настройки обслуживания запросов; каждый из них определяется интерфейсом, который требуется реализовать, или базовым классом, от которого необходимо наследовать.

Один из примеров настройки MVC Framework был продемонстрирован при разработке приложения SportsStore. Мы унаследовали от `DefaultControllerFactory` класс `NinjectControllerFactory`, так что получили возможность создавать контроллеры с использованием Ninject для управления внедрением зависимости (Dependency Injection — DI). Если бы мы следовали этому подходу для каждой точки настройки в Framework MVC, то в конечном итоге смогли бы использовать DI по всему приложению, но тогда получилось бы много дублирования кода и больше обращений к ядру Ninject, чем желательно в идеальном случае.

Когда платформа MVC Framework должна создать экземпляр класса, она вызывает статические методы класса `System.Web.Mvc.DependencyResolver`. Мы можем добавить DI повсюду в приложении MVC, реализовав интерфейс `IDependencyResolver` и зарегистрировав эту реализацию с помощью `DependencyResolver`. В этом случае всякий раз, когда платформа должна создать экземпляр класса, она будет обращаться к нашему классу, и мы сможем вызвать Ninject для создания объекта.

Мы не консолидировали DI при разработке SportsStore, потому что просто хотели продемонстрировать добавление DI к контроллерам. В листинге 10.7 показано, как можно реализовать интерфейс `IDependencyResolver` для этого приложения.

**Листинг 10.7. Реализация Ninject-интерфейса `IDependencyResolver`**

```
using System;
using System.Collections.Generic;
using System.Web.Mvc;
using Ninject;
using Ninject.Parameters;
using Ninject.Syntax;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Concrete;
using SportsStore.WebUI.Infrastructure.Abstract;
```

```
using SportsStore.WebUI.Infrastructure.Concrete;
using System.Configuration;

namespace SportsStore.WebUI.Infrastructure {
    public class NinjectDependencyResolver : IDependencyResolver {
        private IKernel kernel;

        public NinjectDependencyResolver() {
            kernel = new StandardKernel();
            AddBindings();
        }

        public object GetService(Type serviceType) {
            return kernel.TryGet(serviceType);
        }

        public IEnumerable<object> GetServices(Type serviceType) {
            return kernel.GetAll(serviceType);
        }

        public IBindingToSyntax<T> Bind<T>() {
            return kernel.Bind<T>();
        }

        public IKernel Kernel {
            get { return kernel; }
        }

        private void AddBindings() {

            // Поместить сюда дополнительные привязки

            Bind<IProductRepository>().To<EFProductRepository>();
            Bind<IAuthProvider>().To<FormsAuthProvider>();

            // Создать объект настроек электронной почты
            EmailSettings emailSettings = new EmailSettings {
                WriteAsFile = bool.Parse(
                  ConfigurationManager.AppSettings["Email.WriteAsFile"] ?? "false")

            };

            Bind<IOrderProcessor>()
                .To<EmailOrderProcessor>()
                .WithConstructorArgument("settings", emailSettings);
        }
    }
}
```

Этот класс проще, чем выглядит. Первые два метода вызываются MVC Framework, когда требуется новый экземпляр класса, и мы просто обращаемся к ядру Ninject, передавая ему этот запрос. Мы добавили метод `Bind`, так что мы можем добавлять привязки извне этого класса. Это строго необязательно, поскольку мы также включили метод `AddBindings`, который вызывается внутри конструктора, как это делалось в классе `NinjectControllerFactory` из главы 7.

Теперь можно удалить класс `NinjectControllerFactory` и зарегистрировать более общий класс `NinjectDependencyResolver` в методе `Application_Start` файла `Global.asax`, как показано в листинге 10.8.

**Листинг 10.8. Регистрация реализации `IDependencyResolver`**

```
protected void Application_Start() {

    AreaRegistration.RegisterAllAreas();

    RegisterGlobalFilters(GlobalFilters.Filters);
    RegisterRoutes(RouteTable.Routes);

    DependencyResolver.SetResolver(new NinjectDependencyResolver());

    ModelBinders.Binders.Add(typeof(Cart), new CartModelBinder());
}
```

Благодаря этим изменениям, Ninject помещается в ядро приложения MVC. Мы по-прежнему можем пользоваться точками расширения в MVC Framework, но это не понадобится, если нужно всего лишь ввести DI в какую-то часть конвейера запросов.

# Резюме

В этой главе мы описали структуру проекта MVC 3 в Visual Studio и взаимодействие его частей. Мы также коснулись двух самых важных характеристик MVC Framework: соглашения и расширяемости. По мере углубления во внутреннее функционирование MVC Framework, в последующих главах мы будем возвращаться к этим темам вновь и вновь.