МУХАМЕДИЕВА Д.Т., ВАРЛАМОВА Л.П,
БАХРОМОВ С.А., ЭЛОВ Б.Б.,
АБДУРАХМАНОВ О.А.

ПРОГРАММНАЯ ИНЖЕНЕРИЯ ASP.NET MVC 5.0

УЧЕБНОЕ ПОСОБИЕ

МУХАМЕДИЕВА Д.Т., ВАРЛАМОВА Л.П,
БАХРОМОВ С.А., ЭЛОВ Б.Б.,
АБДУРАХМАНОВ О.А.

ПРОГРАММНАЯ ИНЖЕНЕРИЯ ASP.NET MVC 5.0

УЧЕБНОЕ ПОСОБИЕ

МИНИСТЕРСТВО ВЫСШЕГО И СРЕДНЕГО СПЕЦИАЛЬНОГО ОБРАЗОВАНИЯ РЕСПУБЛИКИ УЗБЕКИСТАН

НАЦИОНАЛЬНЫЙ УНИВЕРСИТЕТ УЗБЕКИСТАНА ИМЕНИ МИРЗО УЛУГБЕКА

Мухамедиева Д.Т., Варламова Л.П.,

Бахромов С.А., О.А.Абдурахманов

Учебное пособие

по курсу

«ПРОГРАММНАЯ ИНЖЕНЕРИЯ»

ASP.NET MVC 5

Ташкент 2023

Учебное пособие по курсу “Программная инженерия” составлено на основе образцовой и рабочей программ для студентов специальности 5330100 – “Информационные системы, математика и программное обеспечение” с целью обучения использованию языков программирования, разработке программного обеспечения и систем управления базами данных. Данный курс читается студентам на протяжении трех семестров, в соответствии с чем пособие включает три части. В третьей части пособия рассматриваются вопросы программирования в среде ASP.NET MVC 5.

Авторы:

Д.Т.Мухамедиева - профессор Национального исследовательского университета «Ташкентский институт инженеров ирригации и механизации сельского хозяйства», д.т.н.

Л.П.Варламова - профессор кафедры вычислительной математики и информационных систем Национального университета Узбекистана имени Мирзо Улугбека, д.т.н.

С.А.Бахромов - доцент кафедры вычислительной математики и информационных систем Национального университета Узбекистана имени Мирзо Улугбека, к.т.н.

Б.Б.Элов - доцент кафедры вычислительной математики и информационных систем Национального университета Узбекистана имени Мирзо Улугбека, к.т.н.

О.А.Абдурахманов - ассистент кафедры вычислительной математики и информационных систем Национального университета Узбекистана имени Мирзо Улугбека

Рецензенты:

Матякубов А.С. Зав. кафедрой “Прикладная математика и компьютерный анализ” Национального университета Узбекистана

Якубов М.С. Профессор кафедры “Информационные технологии” Ташкентского университета информационных технологий

Учебное пособие рекомендовано к изданию на основании приказа Министерства высшего и среднего специального образования Республики Узбекистан от 19 июля 2022 года №233. Регистрационный номер 233-0797.

Содержание

Введение в ASP.NET MVC 5

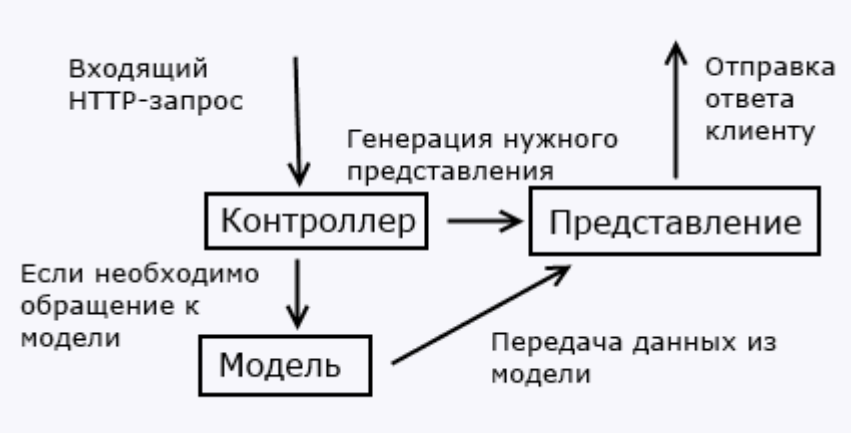
Особенности ASP.NET MVC. Что нового в MVC 5

Платформа **ASP.NET MVC** представляет собой фреймворк для создания сайтов и веб-приложений с помощью реализации паттерна **MVC**.

Концепция паттерна (шаблона) **MVC (model - view - controller)** предполагает разделение приложения на три компонента:

- **Контроллер** (**controller**) представляет класс, обеспечивающий связь между пользователем и системой, представлением и хранилищем данных. Он получает вводимые пользователем данные и обрабатывает их. И в зависимости от результатов обработки отправляет пользователю определенный вывод, например, в виде представления.
- **Представление** (**view**) - это собственно визуальная часть или пользовательский интерфейс приложения. Как правило, **html**-страница, которую пользователь видит, зайдя на сайт.
- **Модель** (**model**) представляет класс, описывающий логику используемых данных.

Общую схему взаимодействия этих компонентов можно представить следующим образом:



В этой схеме модель является независимым компонентом - любые изменения контроллера или представления не затрагивают модель. Контроллер и представление являются относительно независимыми компонентами, и нередко их можно изменять независимо друг от друга.

Благодаря этому реализуется концепция **разделение ответственности**, в связи с чем легче построить работу над отдельными компонентами. Кроме того, вследствие этого приложение обладает лучшей тестируемостью. И если нам, допустим, важная визуальная часть или фронтэнд, то мы можем тестировать представление независимо от контроллера. Либо мы можем сосредоточиться на бэкэнде и тестировать контроллер.

Конкретные реализации и определения данного паттерна могут отличаться, но в силу своей гибкости и простоты он стал очень популярным в последнее время, особенно в сфере веб-разработки.

Свою реализацию паттерна представляет платформа **ASP.NET MVC**. **2013** год ознаменовался выходом новой версии **ASP.NET MVC - MVC 5**, а также релизом **Visual Studio 2013**, которая предоставляет инструментарий для работы с **MVC5**.

Хотя во многих аспектах **MVC 5** не слишком сильно будет отличаться от **MVC 4**, многое из одной версии вполне применимо к другой, но в то же время есть и существенные отличия:

- В **MVC 5** изменилась концепция аутентификации и авторизации. Вместо **SimpleMembershipProvider** была внедрена система **ASP.NET Identity**, которая использует компоненты **OWIN** и **Katana**;
- Для создания адаптивного и расширяемого интерфейса в **MVC 5** используется **css**-фреймворк **Bootstrap;**
- Добавлены фильтры аутентификации, а также появилась функциональность переопределения фильтров;
- В **MVC 5** также добавлены атрибуты маршрутизации.

Это наиболее важные нововведения в **MVC 5**. Кроме того, есть еще ряд менее значимых, например, использование по умолчанию **Entity Framework 6**, некоторые изменения при создании проекта (концепция **One ASP.NET**), дополнительные компоненты и т.д.

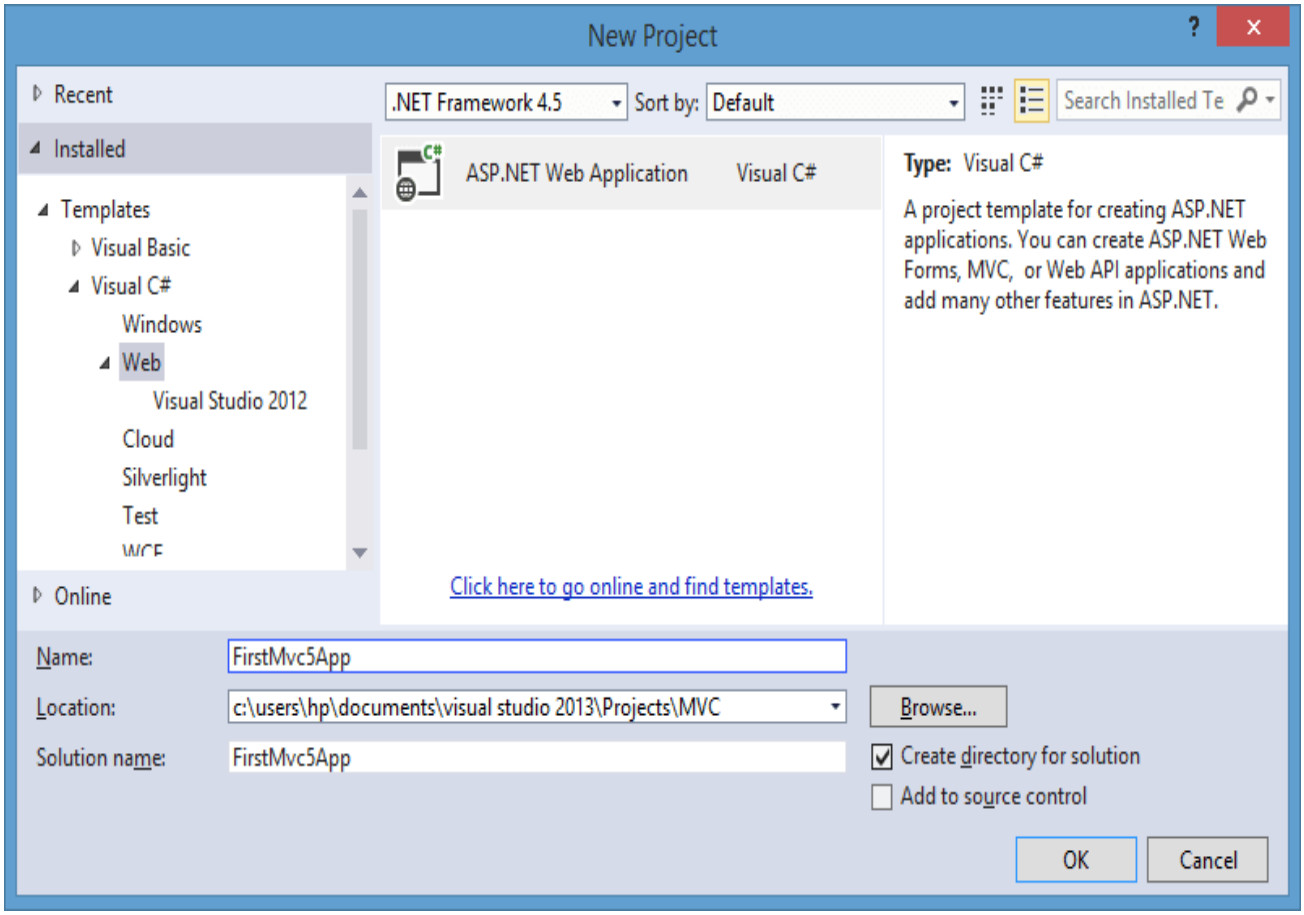
В любом случае все полученные при работе с **MVC 4** навыки можно успешно применять при использовании **MVC 5**, учитывая, конечно, нововведения.

Начало работы с ASP.NET MVC 5

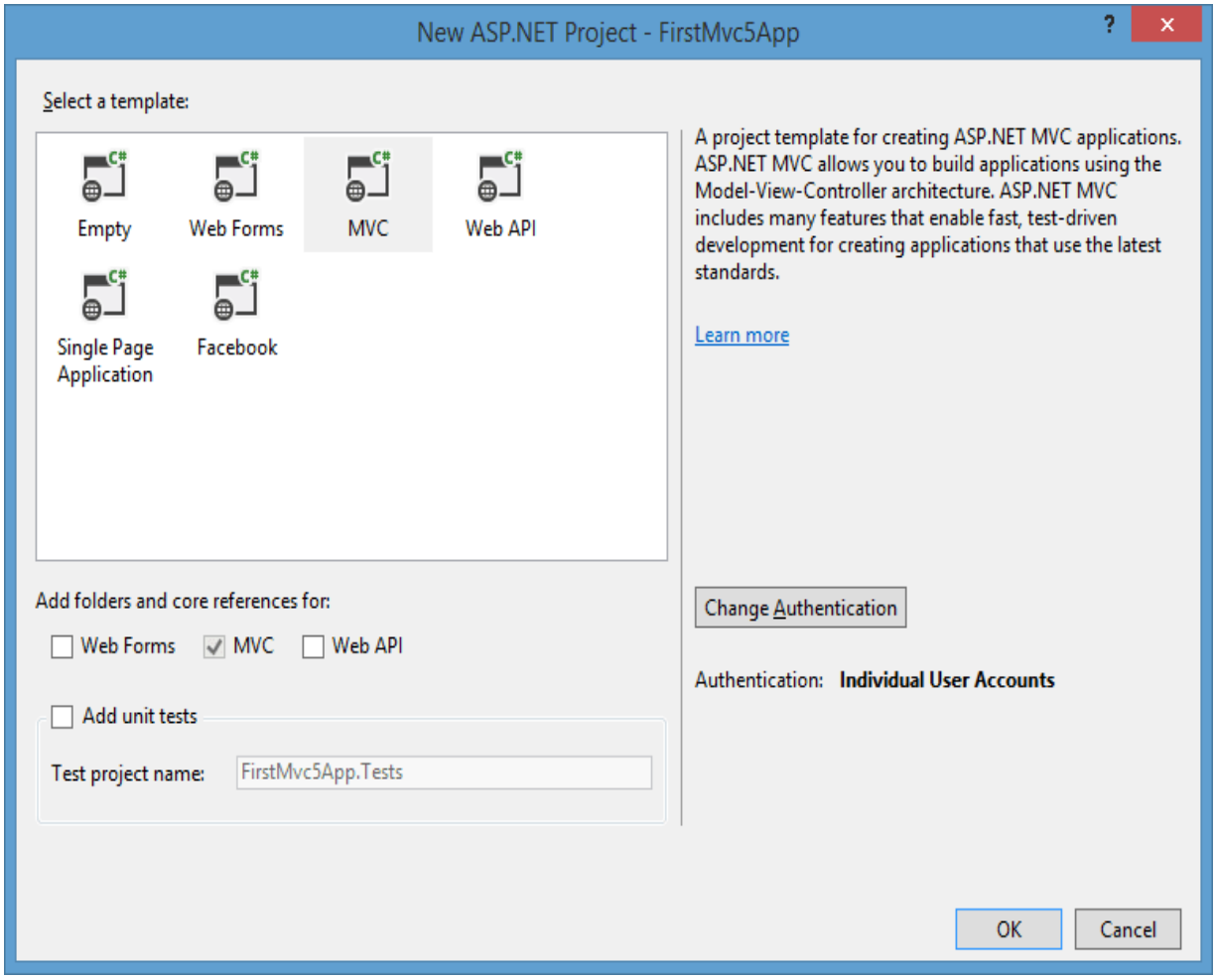
Для создания веб-приложений на платформе **ASP.NET MVC 5** необходима среда разработки - **Visual Studio Express 2013 for Web** (либо другой выпуск **Visual Studio 2013**), которую можно найти по адресу **Visual Studio Express 2013 for Web**. Также существует возможность использовать **MVC 5** и в **Visual Studio 2012**, однако в этом случае надо предварительно установить дополнительный инструментарий - **ASP.NET и Web Tools 2013.1 для Visual Studio 2012**. Я в дальнейшем буду ориентироваться на **Visual Studio 2013**.

После установки откроем **Microsoft Visual Studio Express 2013 for Web** и в меню **File (Файл)** выберем пункт **New Project... (Создать проект)**. Перед нами откроется диалоговое окно создания проекта. Поскольку в компании **Microsoft** взят курс под названием "**One ASP.NET**", то мы не увидим, как в прежних

выпусках **Visual Studio**, разнообразие типов проектов. Вместо этого нам будет доступен только один тип проекта:

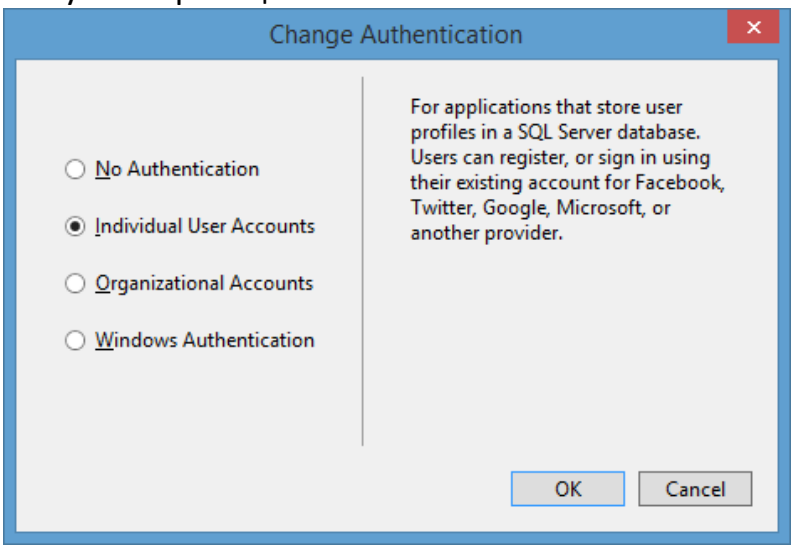


Дадим какое-нибудь имя проекту и нажмем **ОК**. После этого отобразится окно выбора шаблона нового приложения:



По умолчанию уже выбран шаблон **MVC**. Кроме того, данное диалоговое окно задать опции тестирования.

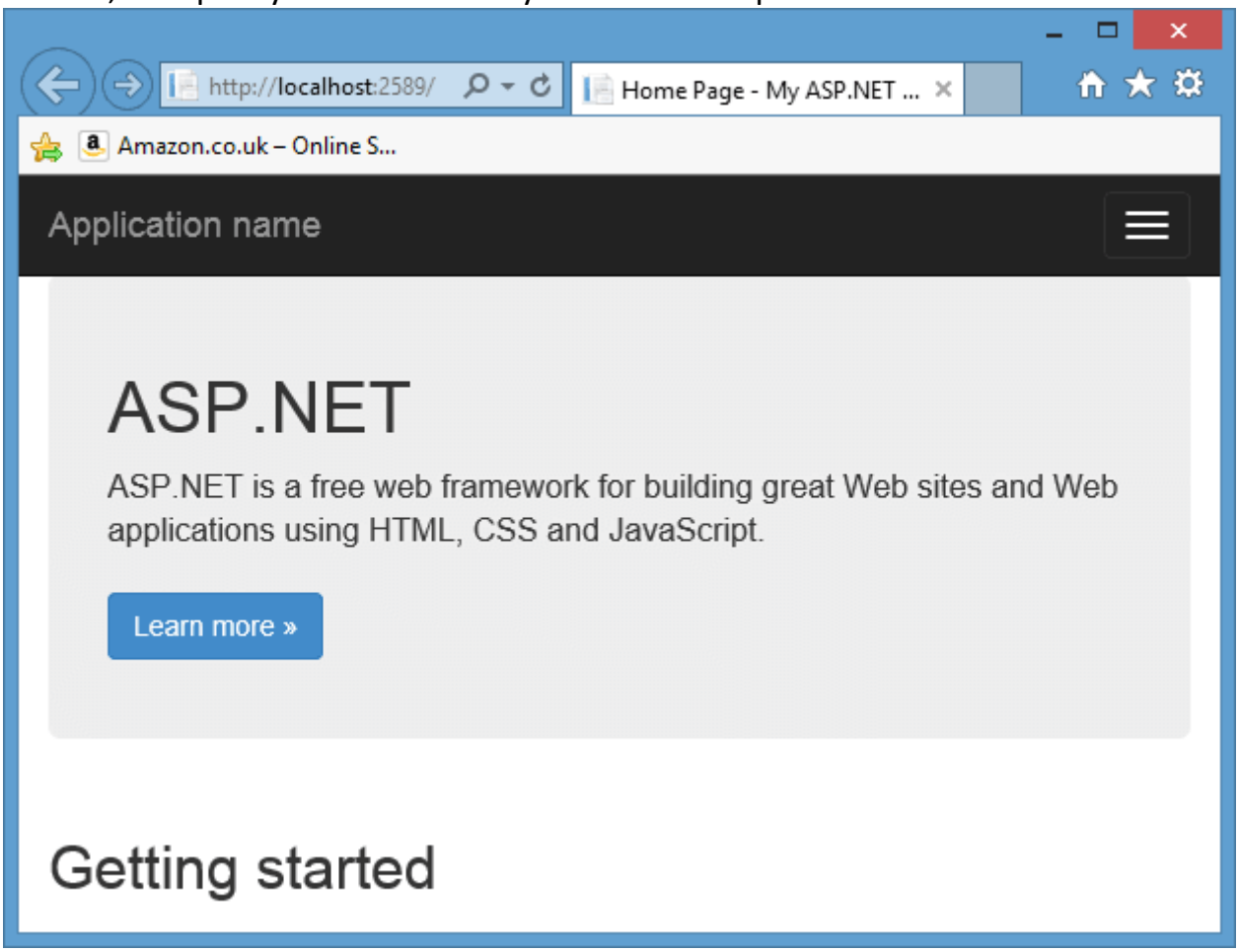
Также нам доступен в правой части окна выбор механизма аутентификации в приложении (кнопка **Change Authentication**). По умолчанию установлен тип **Individual User Accounts**. Не будем его изменять. Но если мы нажмем на кнопку **Change Authentication**, то нам будет доступен выбор из следующих типов аутентификации:



Что они представляют?

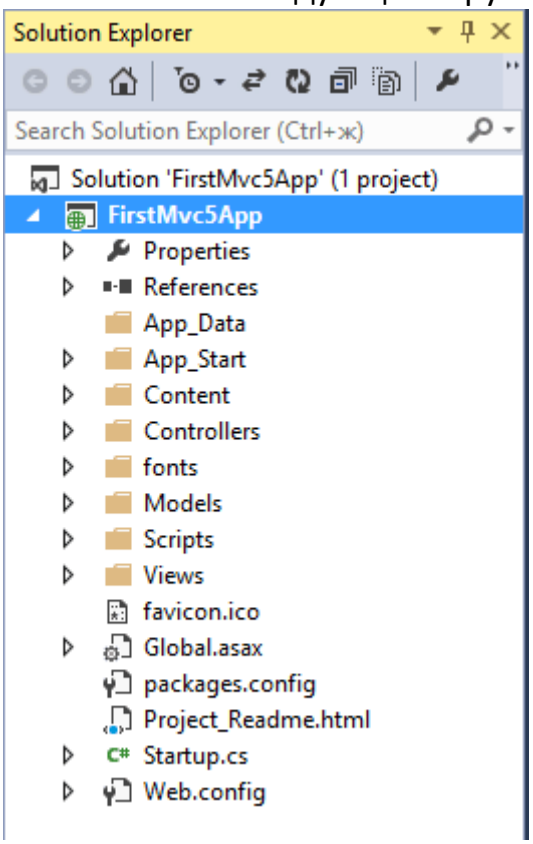
- **No Authentication**: приложение не требует аутентификации пользователя;
- **Individual User Accounts**: требуется индивидуальная аутентификация, учетные записи пользователей хранятся в базе данных, кроме того, доступна аутентификация с помощью социальных сетей;
- **Organizational Accounts**: в основном предназначено для корпоративных приложений, которые используют Active Directory или Office 365;
- **Windows Authentication**: для управления учетными записями используется аутентификация Windows, так называемые intranet-приложения.

Позже мы подробнее поговорим о механизмах аутентификации в приложении. Нажимаем кнопку **OK**, и создается новый проект. Он уже содержит разветвленную структуру и имеет некоторое наполнение по умолчанию. Запустим проект на выполнение, и нам отобразится некоторый контент, который уже имеется по умолчанию в приложении:



Структура проекта MVC 5

Весь этот функционал обеспечивается следующей структурой проекта:



Вкратце рассмотрим, для чего нужны все эти папки и файлы.

- **App_Data**: содержит файлы, ресурсы и базы данных, используемые приложением;
- **App_Start**: хранит ряд статических файлов, которые содержат логику инициализации приложения при запуске;
- **Content**: содержит вспомогательные файлы, которые не включают код на **C#** или **javascript**, и которые развертываются вместе с приложением, например, файлы стилей **css;**
- **Controllers**: содежит файлы классов контроллеров. По умолчанию в эту папку добавляются два контроллера - **HomeController** и **AccountController**;
- **fonts**: хранит дополнительные файлы шрифтов, используемых приложением;
- **Models**: содержит файлы моделей. По умолчанию **Visual Studio** добавляет пару моделей, описывающих учетную запись и служащих для аутентификации пользователя;
- **Scripts**: каталог со скриптами и библиотеками на языке **javascript;**

- **Views**: здесь хранятся представления. Все представления группируются по папкам, каждая из которых соответствует одному контроллеру. После обработки запроса контроллер отправляет одно из этих представлений клиенту. Также здесь имеется каталог **Shared**, который содержит общие для всех представления;
- **Global.asax**: файл, запускающийся при старте приложения и выполняющий начальную инициализацию. Как правило, здесь срабатывают методы классов, определенных в папке **App_Start**;
- **Startup.cs**: поскольку в приложении **MVC 5** используются библиотеки, применяющие спецификацию **OWIN**, то данный файл организует связь между **OWIN** и приложением. (**OWIN** представляет спецификацию, описывающую взаимодействие между компонентами приложения)
- **Web.config**: файл конфигурации приложения.

Конкретная структура каждого отдельного приложения, естественно, будет отличаться, а гибкость **MVC** позволяет изменять структуру, приспосабливая, ее к своим потребностям. Но описанные выше моменты будут общими для большинства проектов. Теперь после ознакомления со структурой проекта создадим первое приложение.

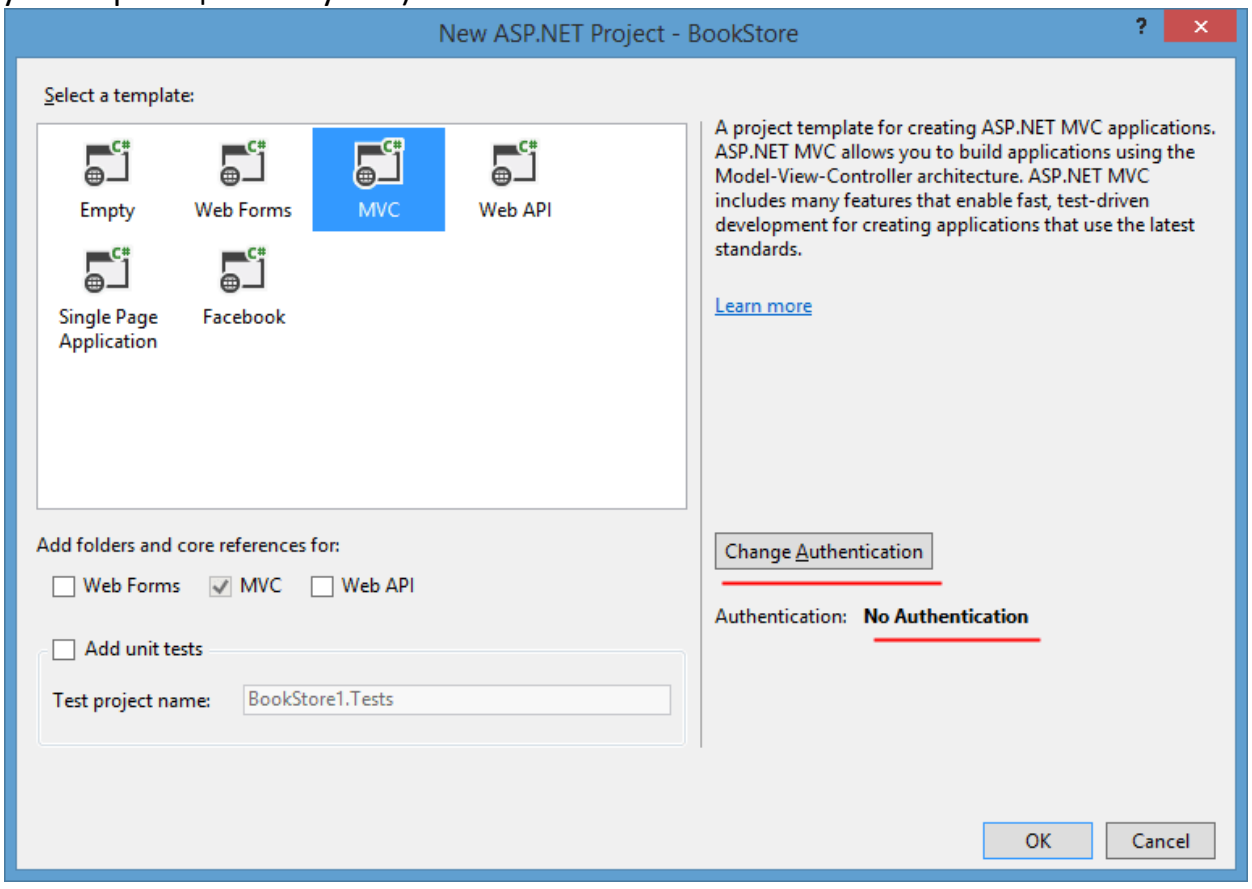
Создание первого приложения ASP.NET MVC 5

Создание проекта

Мы посмотрели некоторые основные понятия паттерна **MVC**, теперь создадим первое приложение. Это будет очень простенькое приложение, цель которого - дать некоторое начальное понимание работы с **ASP.NET MVC 5**.

Это приложение будет эмулировать работ книжного магазина: оно будет предоставлять нам выбор книг, а пользователь, зашедший на сайт, сможет оформить покупку. Для начала, я думаю, достаточно.

Итак, откроем **Visual Studio 2013 File -> New Project..** и создадим новый проект. Назовем новый проект, например, **BookStore**. Затем в окне создания нового проекта выберем **MVC**. А в парвой части окна изменим тип аутентификации приложения на **No Authentication** (так как пока нам система аутентификации не нужна):



После этого будет создан по сути проект, который практически не обладает никакой функциональностью, хотя уже имеет базовую структуру.

Первым делом определим модели данных нашего приложения. Поскольку речь идет о книжном магазине, то такими моделями могут быть модель книги и модель покупки книги.

В проекте уже по умолчанию определена папка **Models**. В ней будут находиться наши модели. Нажмем на эту папку правой кнопкой мыши и в появившемся меню выберем **Add->Class...**. Назовем первый новый класс **Book** и добавим в него код, описывающий модель книги:

```
namespace BookStore.Models
{
    public class Book
    {
        // китоб ID си
        public int Id { get; set; }
        // китоб номи
        public string Name { get; set; }
        // китоб муаллифи
        public string Author { get; set; }
        // нархи
        public int Price { get; set; }
    }
}
```

Конечно, в реальном приложении подобная модель могла бы также включать изображение книги, количество экземпляров, год издания и т.д. Но для нашей задачи вполне сойдет и этот набор полей.

Подобным образом второй класс - модель **Purchase**, которая будет отвечать за отдельную совершаемую покупку книги:

```
using System;
namespace BookStore.Models
{
    public class Purchase
    {
        // харид ID си
        public int PurchaseId { get; set; }
        // харидор исми ва фамилияси
        public string Person { get; set; }
        // харидор манзили
        public string Address { get; set; }
        // китоб ID си
        public int BookId { get; set; }
        // хариф санаси
        public DateTime Date { get; set; }
    }
```

```
}
```

Условности при создании моделей

Как вы видите, модель представляет обычный класс на языке **C#**. Все модели здесь имеют набор свойств, описывающие реальные свойства объекта. В то же время при создании моделей следует соблюдать некоторые условности. Поскольку мы будем использовать для хранения моделей базу данных **SQL Server**, то для манипуляции над объектами в базе данных нам надо определить для них первичный ключ (**Primary Key**), который выполняет роль универсального идентификатора объекта. Поэтому первым свойством в каждой модели идет свойство **Id**, предназначенное для хранения первичного ключа.

И тут вступают в силу условности: свойство идентификатора модели должна иметь имя либо **Имя_моделиId**, либо просто **Id**. Так, у нас в модели **Book** определено свойство **Id**, то есть данное свойство является первичным ключом. А в случае с моделью **Purchase** свойство носит название **PurchaseId**.

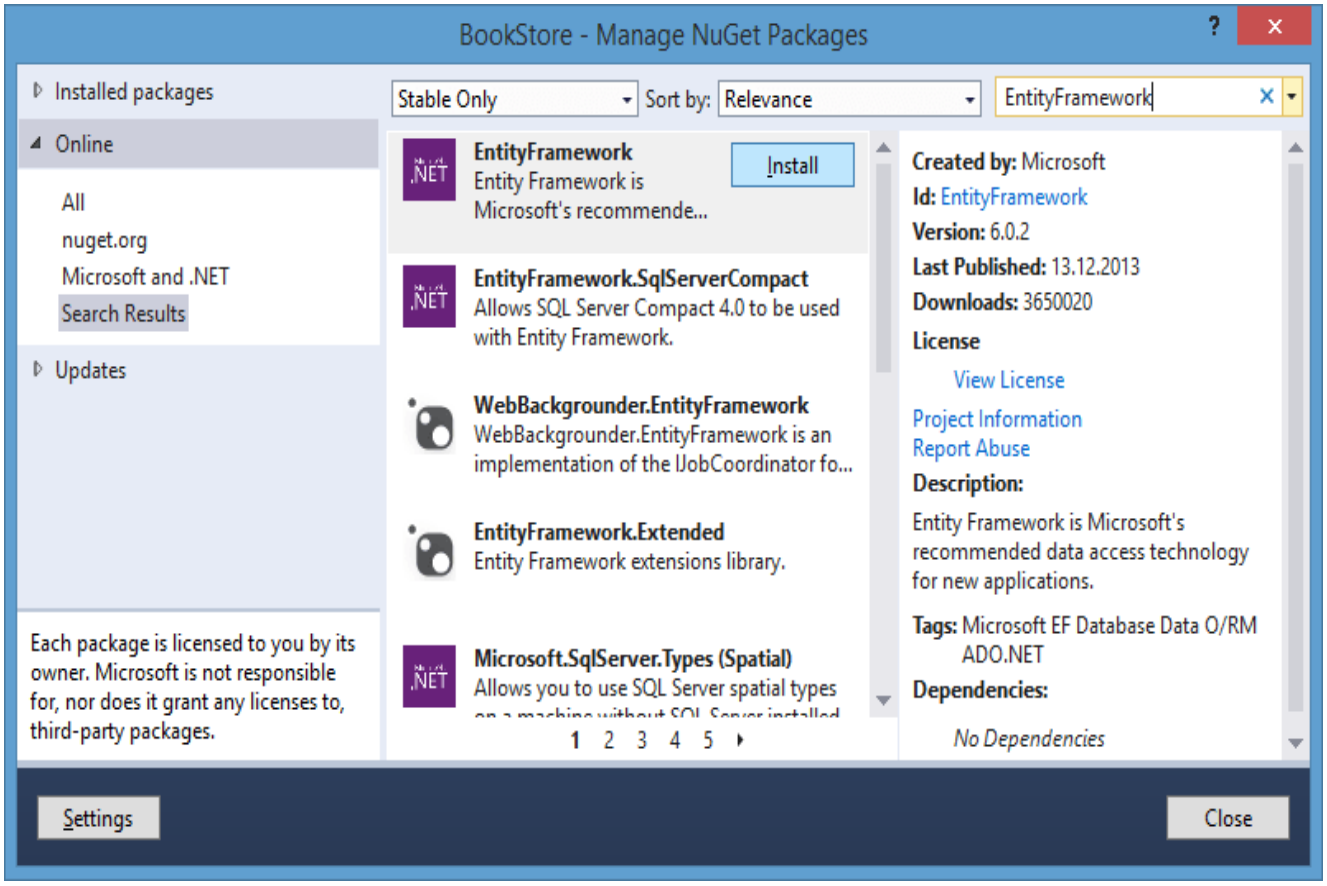
Второй способ состоял в определении ключа с помощью атрибута **Key**, установленным над нужным свойством.

Entity Framework

Для работы с данными в **ASP.NET MVC** рекомендуется использовать фреймворк **Entity Framework**, хотя его использование необязательно и всецело зависит от предпочтений разработчика. Преимущество этого фреймворка состоит в том, что он позволяет абстрагироваться от структуры конкретной базы данных и вести все операции с данными через модель.

Сейчас наш проект не содержит библиотек **Entity Framework**. И чтобы их добавить в проект, воспользуемся пакетным менеджером **NuGet**. Итак, окне **Solution Explorer (Обозреватель решений)** нажмем правой кнопкой мыши в структуре проекта на узел **References** и в появившемся меню выберем **Manage NuGet Packages...**

В окне управления пакетами **NuGet** в правом верхнем углу введите в поле поиска **Entity Framework** и нажмите **Enter**. После этого в среднем столбце будут отображены все найденные пакеты, которые имеют отношение к запросу, а самым первым будет пакет самого фреймворка **Entity Framework**, который нам и надо установить:



Запустим процесс установки пакета, нажав на кнопку **Install**.

Создание контекста данных

После завершения установки создадим контекст данных. Контекст данных использует **Entity Framework** для доступа к **БД** на основе некоторой модели. Итак, добавим в папку **Models** новый класс **BookContext**:

```
using System;
using System.Collections.Generic;
using System.Web;
using System.Data.Entity;

namespace BookStore.Models
{
    public class BookContext : DbContext
    {
        public DbSet<Book> Books { get; set; }
        public DbSet<Purchase> Purchases { get; set; }
    }
}
```

Чтобы создать контекст, нам надо унаследовать новый класс от класса **DbContext**. Свойства наподобие

```
public DbSet<Book> Books { get; set; }
```

помогают получать из **БД** набор данных определенного типа (например, набор объектов **Book**).

Code First

Хотя мы будем использовать базу данных, но создавать явным образом мы ее не будем. За нас все сделает **Entity Framework**. Это так называемый подход **Code First** - у нас есть модели, и по ним фреймворк будет создавать таблицы в базе данных.

И в заключении работы над модельной частью установим строку подключения. Для этого откроем файл **web.config**, найдем секцию **configSections** и сразу **после** нее вставим секцию **connectionStrings**:

```
<configSections>
    <connectionStrings>
      <add name="BookContext" connectionString="Data
Source=(LocalDB)\v11.0;AttachDbFilename='|DataDirectory|\Bookstore.mdf';I
ntegrated Security=True"
   providerName="System.Data.SqlClient" />
    </connectionStrings>
</configSections>
```

В этой секции мы определяем путь к базе данных, которая затем будет создаваться. Выражение **|DataDirectory|** представляет заместитель, который указывает, что база данных будет создаваться в проекте в папке **App_Data**. Позднее мы подробнее разберем настройки подключения к **БД**.

Создание контроллера и представлений

Так как с моделями и настройкой контекста данных мы закончили, то займемся другим компонентом приложения - контроллером. Для контроллеров предназначена папка **Controllers**. По умолчанию при создании проекта в нее добавляется контроллер **HomeController**, который практически не имеет никакой функциональности, и сейчас его код выглядит следующим образом:

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

```
namespace BookStore.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }

        public ActionResult About()
        {
            ViewBag.Message = "Your application description page.";

            return View();
        }

        public ActionResult Contact()
        {
            ViewBag.Message = "Your contact page.";

            return View();
        }
    }
}
```

В контроллере определены по умолчанию три метода: **Index, About** и **Contact**. Нам они не нужны. Изменим код контроллера на следующий:

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

namespace BookStore.Controllers
{
    public class HomeController : Controller
    {
        // создаем контекст данных
        BookContext db = new BookContext();

        public ActionResult Index()
        {
            // получаем из бд все объекты Book
            IEnumerable<Book> books = db.Books;
```

```
            // передаем все объекты в динамическое свойство Books в
ViewBag
            ViewBag.Books = books;
            // возвращаем представление
            return View();
        }
    }
}
```

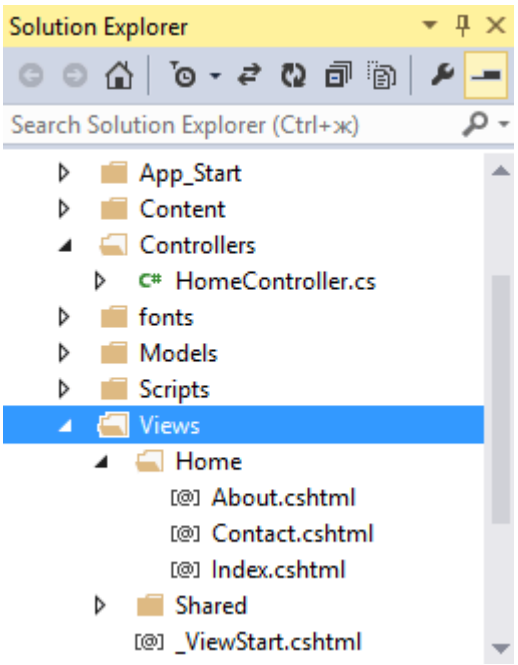
Прежде всего, мы подключаем пространство имен моделей, даже не смотря на то, что он находятся в одном проекте, но в разных пространствах. Затем создается объект контекста данных, через который мы будем взаимодействовать с **БД**:

```
BookContext db = new BookContext();
```

Далее используя метод **db.Books**, получаем из базы данных набор объектов **Book**. Теперь надо передать этот набор в представление.

Для передачи списка объектов **Book** в представлением используем объект **ViewBag**. **ViewBag** представляет такой объект, который позволяет определить любую переменную и передать ей некоторое значение, а затем в представлении извлечь это значение. Так, мы определяем переменную **ViewBag.Books**, которая и будет хранить набор книг.

Теперь создадим само представление для вывода списка книг. Для представлений в проекте предназначена папка **Views**. По умолчанию в этой папке уже есть подкаталог для представлений контроллера **Home**, в котором три представления: **About.cshtml, Contact.cshtml** и **Index.chtml**.



Первые два представления нам уже не понадобятся, и их можно спокойно удалить. А представление **Index.cshtml** откроем и изменим следующим образом:

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Книжный магазин</title>
</head>
<body>
    <div>
        <h3>Распродажа книг</h3>
        <table>
            <tr>
                <td><p>Название книги</p></td>
                <td><p>Автор</p></td>
                <td><p>Цена</p></td>
                <td></td>
            </tr>
            @foreach (var b in ViewBag.Books)
            {
                <tr>
                    <td><p>@b.Name</p></td>
                    <td><p>@b.Author</p></td>
                    <td><p>@b.Price</p></td>
                    <td><p><a href="/Home/Buy/@b.Id">Купить</a></p></td>
                </tr>
            }
        </table>
    </div>
</body>
</html>
```

Самым первым выражением **Layout = null**; мы указываем, что мастер-страница не будет применяться к этому представлению. Далее мы добавим к нему мастер-страницу и узнаем, зачем она нужна, а пока обойдемся без нее.

Практически весь остальной код представляет собой стандартный код на языке **html**: создание обычной таблицы, которая выводит информацию о продаваемых книгах. Здесь также используется интересная конструкция `@foreach (var b in ViewBag.Books)`.

Эта конструкция применяет синтаксис **Razor**. Подробнее о движке **Razor** и его ситаксисе мы поговорим в отдельной главе, а пока вам надо знать, что после символа **@** согласно синтаксису мы можем использовать выражения кода на языке **C#/VB.NET**.

То есть тут мы создаем цикл. В нем мы пробегаемся по всем элементам в объекте **ViewBag.Books**, который был ранее создан в методе контроллера. И затем получаем значение свойства каждого элемента с помощью синтаксиса **Razor: @b.Name** и помещаем его в ячейку таблицы.

В последнюю колонку таблицы для каждого элемента добавляется ссылка `<a href="/Home/Buy/@b.Id">Купить</a>`. При нажатии на эту ссылку методу **Buy** контроллера **HomeController** будет отправляться запрос, в котором вместо **@b.Id** будет указан **id** книги. Пока у нас, правда, отсутствует метод методу **Buy**, но скоро мы его создадим.

Основы маршрутизации

Чтобы обратиться к контроллеру **HomeController** или отправить ему запрос, нам надо указать в строке запроса его имя - **Home**. Кроме того, после имени контроллера нам надо через слеш указать действие или метод контроллера, к которому отправляется запрос. По умолчанию при запуске проекта или при обращении к сайту система **mvc** будет вызывать действие **Index** контроллера **HomeController**, если мы не укажем иной маршрут по умолчанию в параметрах маршрутизации.

Путь **/Home/Buy** означает, что мы будем обращаться к методу **Buy** контроллера **HomeController**. А добавление в запрос параметра **/Home/Buy/@b.Id**, означает, что такой метод может принимать параметр. Но перед тем как создать этот метод, наполним приложение данными.

Данные для моделей по умолчанию

Так как мы будем использовать подход **Code First**, то нам не надо вручную создавать базу данных и наполнять ее данными. Мы можем воспользоваться специальным классом, который за нас добавит начальные данные в **БД**. Для этого в папку **Models** добавим новый класс `BookDbInitializer` и изменим его код следующим образом:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data.Entity;
```

```
namespace BookStore.Models
{
    public class BookDbInitializer :
DropCreateDatabaseAlways<BookContext>
    {
        protected override void Seed(BookContext db)
        {
            db.Books.Add(new Book { Name = "Война и мир", Author = "Л.
Толстой", Price = 220 });
            db.Books.Add(new Book { Name = "Отцы и дети", Author = "И.
Тургенев", Price = 180 });
            db.Books.Add(new Book { Name = "Чайка", Author = "А. Чехов",
Price = 150 });

            base.Seed(db);
        }
    }
}
```

Класс **DropCreateDatabaseAlways** позволяет при каждом новом запуске заполнять базу данных заново некоторыми начальными данными. В качестве таких начальных значений здесь создаются три объекта **Book**.

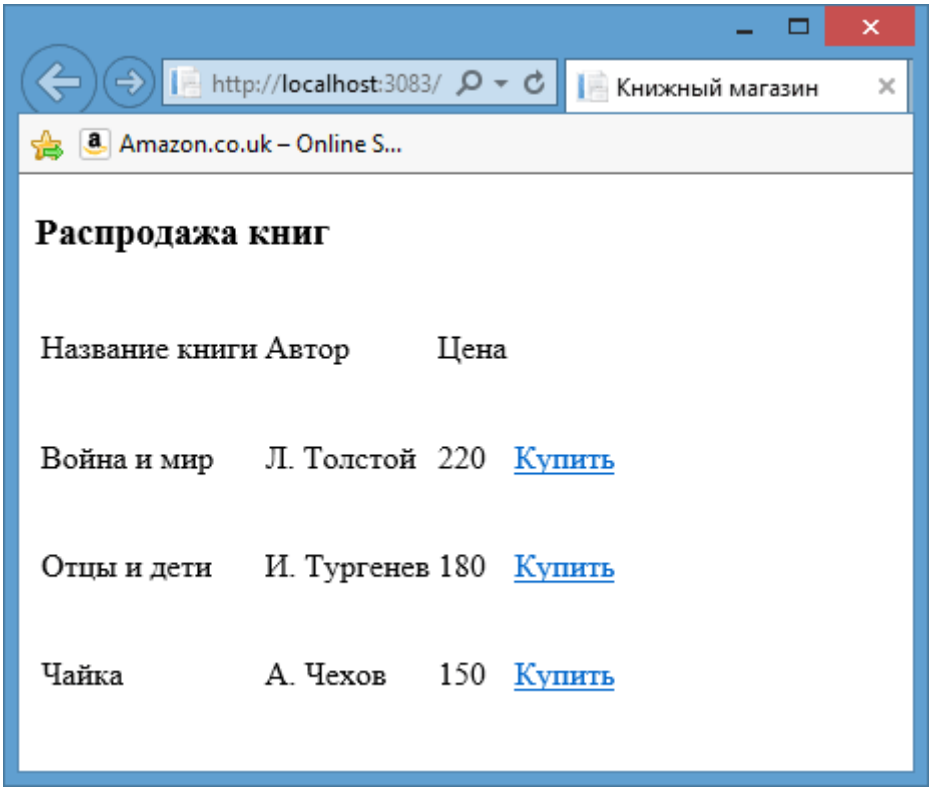
Используя метод **db.Books.Add** мы добавляем каждый такой объект в базу данных.

Однако чтобы этот класс действительно сработал, и заполнение базы данных произошло, нам надо запустить его при запуске приложения. Все начальные настройки приложения и конфигурации находятся в файле **Global.asax**. Откроем его и добавим в метод **Application_Start**, который отрабатывает при старте приложения, следующую строку **Database.SetInitializer(new BookDbInitializer());**:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Optimization;
using System.Web.Routing;
using BookStore.Models;
using System.Data.Entity;

namespace BookStore
{
    public class MvcApplication : System.Web.HttpApplication
```

```
    {
        protected void Application_Start()
        {
            Database.SetInitializer(new BookDbInitializer());

            AreaRegistration.RegisterAllAreas();
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);
        }
    }
}
```

И, наконец, мы можем запустить проект на выполнение и увидеть на веб-странице в браузере наши данные по умолчанию:



И если мы откроем папку проекта (не в **Visual Studio**, а на жестком диске) и зайдем в ней в каталог **App_Data**, то можем увидеть только что созданную базу данных **Bookstore.mdf**, которая и хранит эти данные по умолчанию.

Теперь же создадим выше обсуждавшийся метод **Buy**, который отвечает за покупку книги. Добавим в контроллер **HomeController** следующие два метода:

```
[HttpGet]
        public ActionResult Buy(int id)
```

```
        {
            ViewBag.BookId = id;
            return View();
        }

[HttpPost]
        public string Buy(Purchase purchase)
        {
            purchase.Date = DateTime.Now;
            // добавляем информацию о покупке в базу данных
            db.Purchases.Add(purchase);
            // сохраняем в бд все изменения
            db.SaveChanges();
            return "Спасибо," + purchase.Person + ", за покупку!";
        }
```

Хотя здесь два метода, но в целом они составляют одно действие **Buy**, только первый метод срабатывает при получении запроса **GET**, а второй - при получении запроса **POST**. С помощью атрибутов **[HttpGet]** и **[HttpPost]** мы можем указать, какой метод какой тип запроса обрабатывает.

Так как предполагается, что в метод **Buy** будет передаваться **id** книги, которую пользователь хочет купить, то нам надо определить в методе соответствующий параметр: **public ActionResult Buy(int id)**.

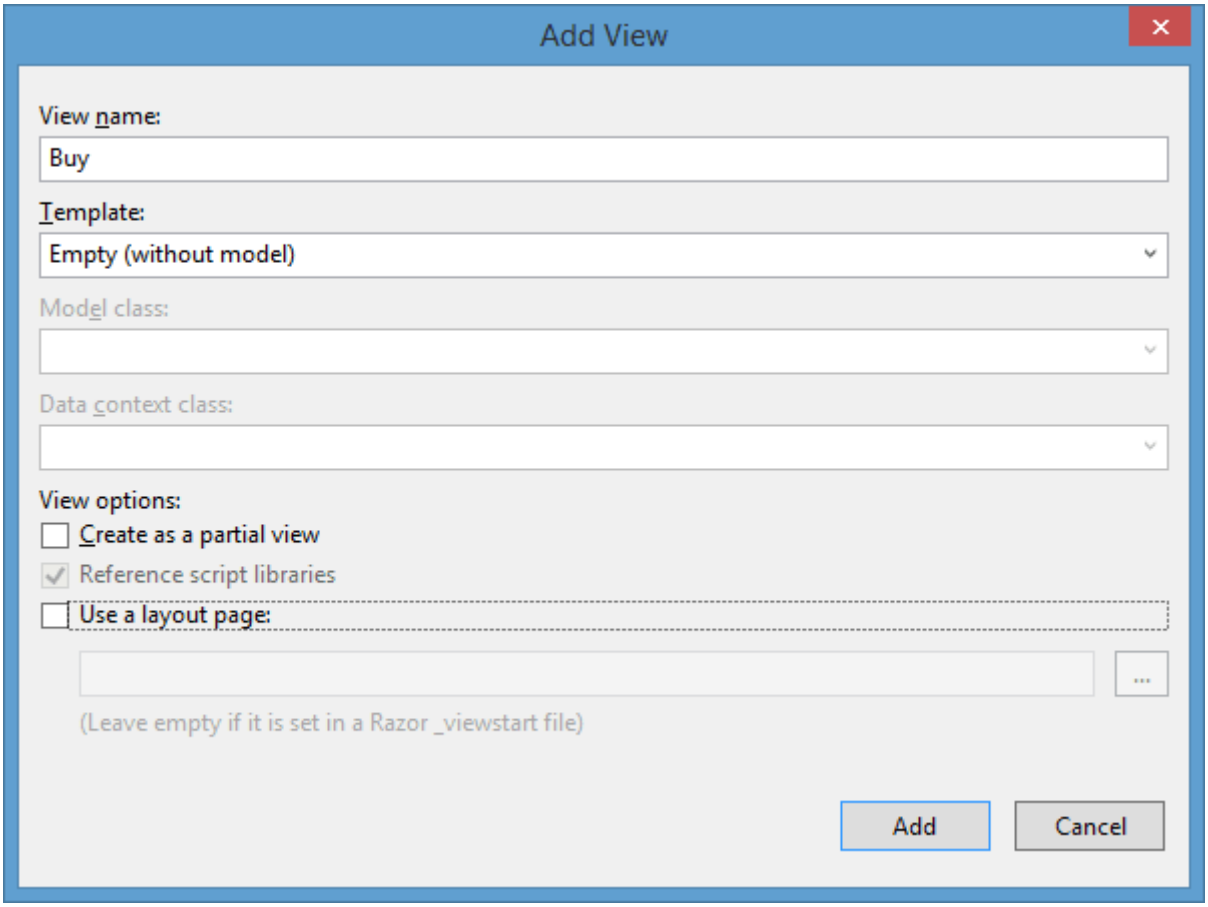
Затем этот параметр передается через объект **ViewBag** в представление, которое мы сейчас создадим.

Метод **public string Buy(Purchase purchase)** выглядит несколько сложнее. Он принимает переданную ему в запросе **POST** модель **purchase** и добавляет ее в базу данных. Результатом работы метода будет строка, которую увидит пользователь.

А весь код по добавлению нового объекта в **БД** благодаря использованию **Entity Framework** фактически сводится к двум строчкам:

```
db.Purchases.Add(purchase);
db.SaveChanges();
```

И в конце добавим представление **Buy.cshtml**. Для этого нажмем на метод **public ActionResult Buy(int id)** правой кнопкой и в появившемся списке выберем **Add View...(Добавить представление)**. Перед нами откроется окно добавления нового представления:



Оставим все установки по умолчанию, только снимем галочку с поля **Use a layout page**, так как пока мастер-страницу мы не будем использовать. И нажмем **Add (Добавить)**. Изменим код нового представления следующим образом:

```
@{
    Layout = null;
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Покупка</title>
</head>
<body>
    <div>
        <h3>Форма оформления покупки</h3>
        <form method="post" action="">
            <input type="hidden" value="@ViewBag.BookId" name="BookId" />
            <table>
                <tr>
```

```
                    <td><p>Введите свое имя </p></td>
                    <td><input type="text" name="Person" /> </td>
                </tr>
                <tr>
                    <td><p>Введите адрес :</p></td>
                    <td>
                        <input type="text" name="Address" />
                    </td>
                </tr>
                <tr><td><input type="submit" value="Отправить" />
</td><td></td></tr>
            </table>
        </form>
    </div>
</body>
</html>
```

При переходе на главной странице по ссылке **"/Home/Buy/2"** контроллер будет получать запрос к действию **Buy**, передавая ему в качестве параметра **id** значение **2**. И так как такой запрос представляет тип **GET**, пользователю будет возвращаться данное представление с формой.

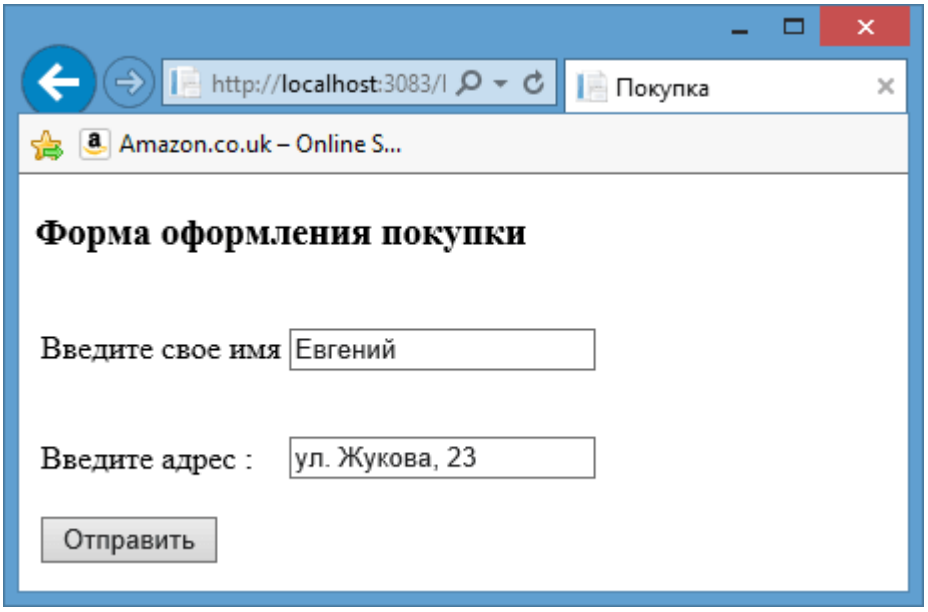
Представление по сути представляет собой форму для ввода данных. Обратите внимание, что так как нам не надо изменять значение **BookId**, однако это значение все равно нам нужно для формирования модели **Purchase**, то мы его вкладываем в скрытое поле в начале формы.

После заполнения формы и нажатия на кнопку форма будет оправляться запросом **POST**, так как мы его определили в строке **<form method="post" action="">**. Контроллер снова будет получать запрос к методу **Buy**, только теперь будет выбираться для обработки запроса метод **public string Buy(Purchase purchase)**.

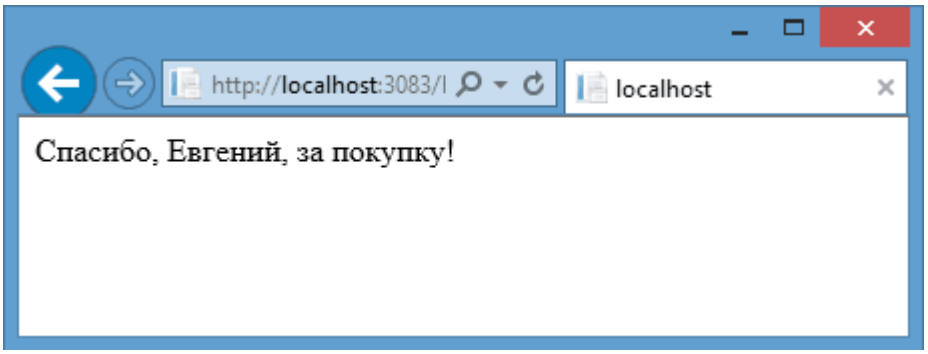
Как система **MVC** угадывает, что мы передали с запросом **post** информацию о модели **Purchase**, а не набор разрозненных значений полей формы? Обратите внимание на поля ввода **<input type="text" name="Person" />** и **<input type="text" name="Address" />**.

Значение их атрибута **name** соответствуют именам свойств модели **Purchase**. После нажатия кнопки и отправки запроса приложение получает значения этих полей. Система **MVC**, используя соглашения по умолчанию, считает эти значения значениями соответствующих свойств модели и проводит связывание отдельных значений со свойствами модели.

Теперь запустим наше приложение. На главной странице выберем какую-нибудь книгу и нажмем на ссылку "**Купить**". На форме заполним поля и нажмем кнопку "**Отправить**".



После нажатия кнопки информация о покупке попадет в базу данных, а в браузер отобразит уведомление:

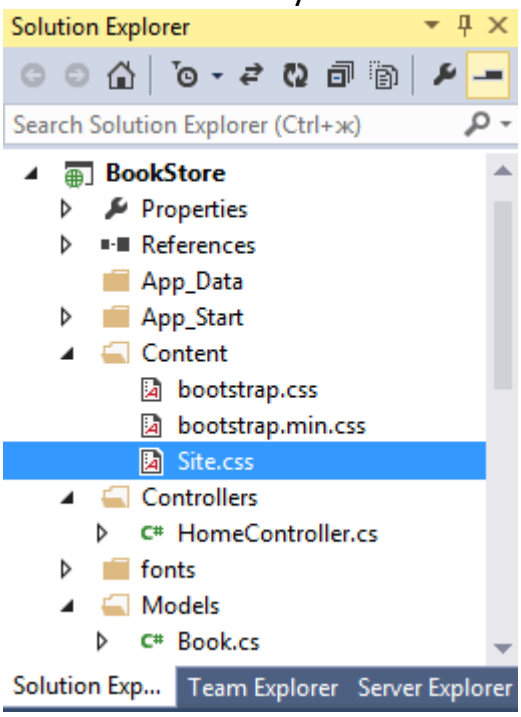


На этом небольшое мини-тренировочное приложение готово. Теперь сделаем приложение красивее, добавив стилизацию. А также добавим поддержку мастер-страниц.

(Видео по материалу: 2.2.Создание контроллера и 2.3.Добавление метода контроллера)

Стилизация приложения и мастер-страницы

Итак, добавим в наше приложение небольшую примитивную стилизацию. Для этого определим файл стилей. По умолчанию **Visual Studio** уже добавляет файл стилей **Site.css** в папку **Content**:



Кроме файла **Site.css**, в папке **Content** находится файл **css**-фреймворка **Bootstrap**, но нам он пока не понадобится. Откроем файл **Site.css** и изменим его содержание на следующее:

```
body {
    font-size: 13px;
    font-family: Verdana, Arial, Helvetica, Sans-Serif;
    background-color: #f7f7fa;
    padding-left:40px;
}

nav{
    display: block;
}

.menu {
    padding-left:10px;
}
.menu ul {
    list-style:none;
}
.menu li {
```

```
    display:inline;
}
.menu a:hover {
    color:red;
}
table {
   vertical-align:middle;
   text-align:left;
 }
.header {
    font-weight:bold;
}
 td {
    padding-right:10px;
 }
input {
    width: 150px;
}
```

Класс **.menu** в данном случае будет служить в качестве класса для навигационного меню на сайте. Хотя наше приложении не очень большое, поэтому там будет только ссылка на главную страницу. Но при необходимости вы можете добавить в него дополнительные пункты.

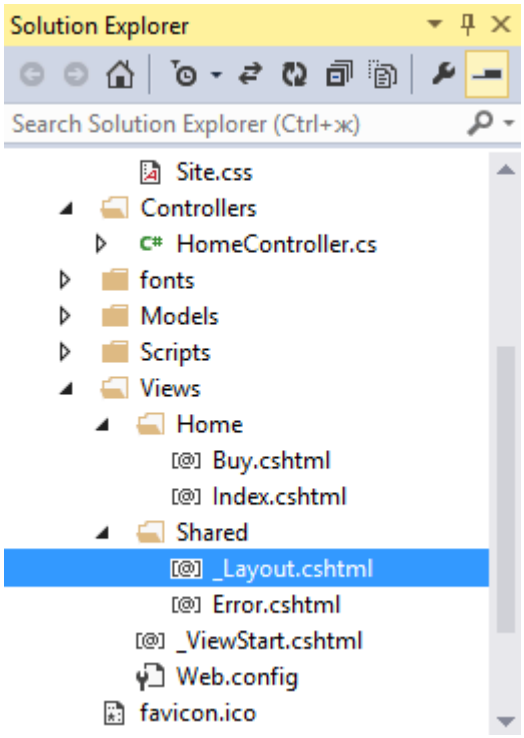
Чтобы использовать стили, мы можем их просто подключить в секции head, как в любой обычной **html**-страницу:

```
<head>
    <meta name="viewport" content="width=device-width" />
    <link type="text/css" rel="stylesheet" href="../../Content/Site.css"
/>
</head>
```

В нашем случае достаточно вставить данный код на оба наших представления. Однако это не самый лучший подход, так как стили для обоих представлений общие, кроме того, подобных представлений в проекте может быть не два, а гораздо больше. И если мы вдруг изменим ссылку на файл стилей, то придется менять эту ссылку на всех представлениях.

И чтобы выйти из этой проблемы фреймворк **ASP.NET MVC** предоставляет нам такую функциональность, как мастер-страницы. Мастер-страница задает единый шаблон для других использующих его представлений.

По умолчанию в проекте уже имеется мастер-страница, которая называется **_Layout.cshtml**. Ее можно найти в папке **Views -> Shared**:



Файл **_Layout.cshtml** уже имеет некоторое содержимое по умолчанию. Изменим его на следующее:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>@ViewBag.Title</title>
    <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet"
type="text/css" />
</head>

<body>
    <nav>
        <ul class="menu">
            <li>@Html.ActionLink("Главная", "Index", "Home")</li>
        </ul>
    </nav>
    @RenderBody()
</body>
</html>
```

Для подключения стилей здесь использовался другой способ - метод **Url.Content**, в который передается путь к файлу. Впоследствии мы познакомимся с еще одним способом - подключение **бандл**ов, который является более предпочтительным.

После секции **head** на мастер-странице идет создание меню. Так как у нас всего два представления, то в качестве одного единственного пункта меню указывается ссылка на главную страницу. Для создания ссылки используется метод **Html.ActionLink**. Он генерирует элемент-ссылку и принимает название ссылки, метод контроллера и имя контроллера.

И далее идет вызов метода **RenderBody()** - с помощью этого метода в это место будет подставляться разметка уже конкретных представлений.

Теперь изменим представления так, чтобы они использовали мастер-страницу. Обновленное представление **Index.cshtml**:

```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<div>
    <h3>Распродажа книг</h3>
    <table>
        <tr class="header">
            <td><p>Название книги</p></td>
            <td><p>Автор</p></td>
            <td><p>Цена</p></td>
            <td></td>
        </tr>
        @foreach (var b in ViewBag.Books)
        {
            <tr>
                <td><p>@b.Name</p></td>
                <td><p>@b.Author</p></td>
                <td><p>@b.Price</p></td>
                <td><p><a href="/Home/Buy/@b.Id">Купить</a></p></td>
            </tr>
        }
    </table>
</div>
```

Также изменим представление **Buy.cshtml**:

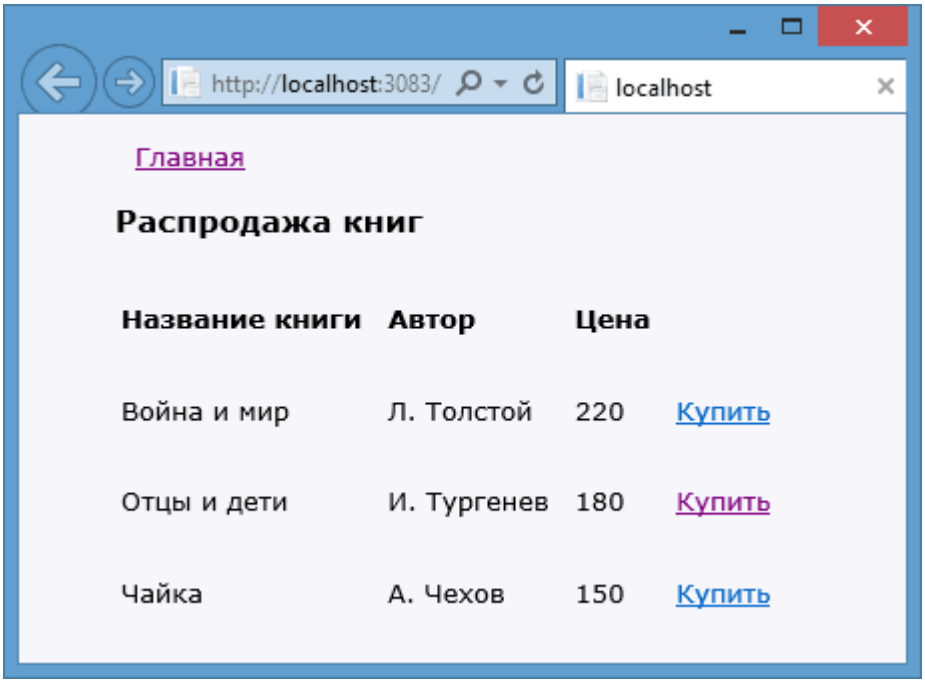
```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<div>
    <h3>Форма оформления покупки</h3>
    <form method="post" action="">
        <input type="hidden" value="@ViewBag.BookId" name="BookId" />
        <table>
            <tr>
```

```
                <td><p>Введите свое имя </p></td>
                <td><input type="text" name="Person" /> </td>
            </tr>
            <tr>
                <td><p>Введите адрес :</p></td>
                <td>
                    <input type="text" name="Address" />
                </td>
            </tr>
            <tr><td><input type="submit" value="Отправить" />
</td><td></td></tr>
        </table>
    </form>
</div>
```

Чтобы указать используемую мастер-страницу, в начале представления прописывается путь к мастер-странице:

```
Layout = "~/Views/Shared/_Layout.cshtml";
```

И теперь нам больше не нужны секции **head** и **body**. Мы их можем удалить. Запустим на выполнение обновленный проект и убедимся, что к нашему приложению применена стилизация и мастер-страницы:



На этом работа над приложением закончена, и теперь мы можем перейти к более детальному обсуждению основных компонентов приложения **MVC**.

Контроллеры

Основы контроллеров

Контроллер является центральным компонентом в архитектуре **MVC**. Контроллер получает ввод пользователя, обрабатывает его и посылает обратно результат обработки, например, в виде представления.

При использовании контроллеров существуют некоторые условности. Так, по соглашениям об именовании названия контроллеров должны оканчиваться на суффикс "**Controller**", остальная же часть до этого префикса считается именем контроллера.

Чтобы обратиться контроллеру из веб-браузера, нам надо в адресной строке набрать **адрес_сайта/Имя_контроллера/**.

Так, по запросу **адрес_сайта/Home/** система маршрутизации по умолчанию вызовет метод **Index** контроллера **HomeController** для обработки входящего запроса. Если мы хотим отправить запрос к конкретному методу контроллера, то нужно указывать этот метод явно: **адрес_сайта/Имя_контроллера/Метод_контроллера**.

Например, **адрес_сайта/Home/Buy** - обращение к методу **Buy** контроллера **HomeController**.

Контроллер представляет обычный класс, который наследуется от базового класса **System.Web.Mvc.Controller**. В свою очередь класс **Controller** реализует абстрактный базовый класс **ControllerBase**, а через него и интерфейс **IController**. Таким образом, формально, чтобы создать свой класс контроллера, достаточно создать класс, реализующий интерфейс **IController** и имеющий в имени суффикс **Controller**.

Интерфейс **IController** определяет один единственный метод **Execute**, который отвечает за обработку контекста запроса:

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

Теперь создадим какой-нибудь простенький контроллер, реализующий данный интерфейс. В качестве проекта мы можем взять проект из предыдущий главы. Итак, добавим в папку **Controllers** проекта новый класс (именно класс, а не контроллер) со следующим содержанием:

```
using System.Web.Mvc;
using System.Web.Routing;
```

```
namespace BookStore.Controllers
{
    public class MyController : Controller
    {
        //
        // GET: /My/
        public void Execute(RequestContext requestContext)
        {
            string ip =
requestContext.HttpContext.Request.UserHostAddress;
            var response = requestContext.HttpContext.Response;
            response.Write("<h2>Ваш IP-адрес: " + ip + "</h2>");
        }
     }
}
```

При обращении к любому контроллеру система передает в него контекст запроса. В этот контекст запроса включается все: куки, отправленные данные форм, строки запроса, идентификационные данные пользователя и т.д. Реализация интерфейса **IController** позволяет получить этот контекст запроса в методе **Execute** через параметр **RequestContext**. В нашем случае мы получаем **IP-адрес** пользователя через свойство **requestContext.HttpContext.Request.UserHostAddress**.

Кроме того, мы можем отправить пользователю ответ с помощью объекта **Response** и его метода **Write**.

Таким образом, перейдя по пути **адрес_сайта/My/**, пользователь увидит свой **IP-адрес**.

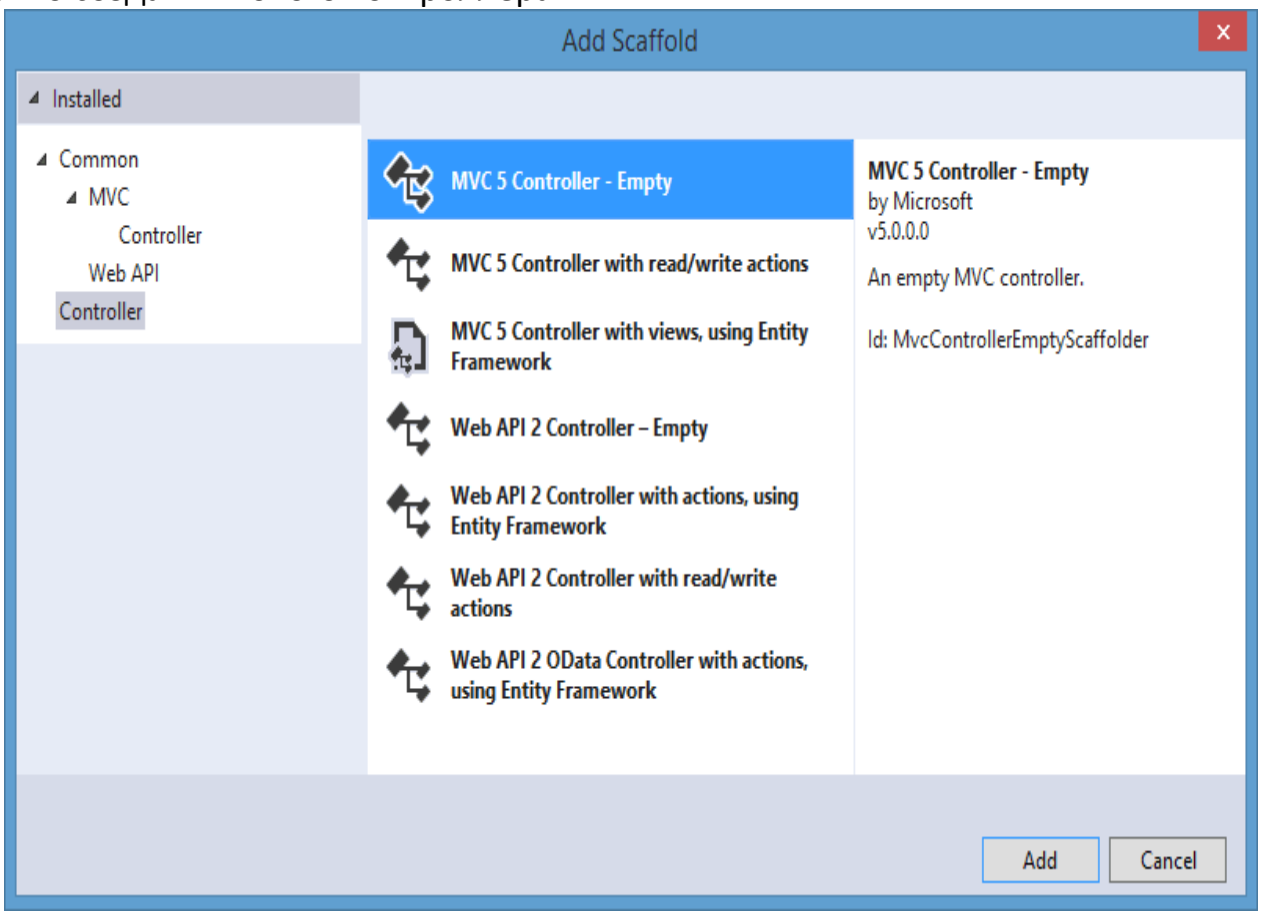
Хотя с помощью реализации интерфейса **IController** очень просто создавать контроллеры, но в реальности чаще оперируют более высокоуровневыми классами, как например класс **Controller**, поскольку он предоставляет более мощные средства для обработки запросов. И если при реализации интерфейса **IController** мы имеем дело с одним методом **Execute**, и все запросы к этому контроллеру, будут обрабатываться только одним методом, то при наследовании класса **Controller** мы можем создавать множество методов действий, которые будут отвечать за обработку входящих запросов, и возвращать различные результаты действий.

Чтобы создать стандартный контроллер, мы можем также добавить в папку **Controllers** простой класс и унаследовать от класса **Controller**, например:

```
using System.Web.Mvc;

namespace BookStore.Controllers
{
```

```
    public class BookShopController : Controller
    {
        //
        // GET: /BookShop/
        public ActionResult Index()
        {
            return View();
        }
     }
}
```

Однако **Visual Studio** предлагает нам более удобные средства для создания контроллеров, предполагающие их гибкую настройку. Чтобы ими воспользоваться, нажмем на папку **Controllers** правой кнопкой мыши и в появившемся меню выберем **Add -> Controller....** После этого нам отобразится окно создания нового контроллера:



Собственно к контроллерам **MVC 5** здесь непосредственное отношение имеют первые три пункта. Остальные больше относятся к **Web API 2**. В этом списке выберем первый пункт - **MVC 5 Controller - Empty**, который подразумевает создание пустого контроллера. Остальные два пункта

позволяют сгенерировать классы с **CRUD**-функциональностью на основе шаблонов формирования, о которых мы поговорим в разделе о моделях.

Далее нам будет предложено ввести имя, и после этого новый контроллер с единственным методом **Index** будет добавлен в проект. При таком добавлении в отличие от предыдущих примеров для данного контроллера будет автоматически создан каталог в папке **Views**, который будет хранить все представления, связанные с действиями этого контроллера.

Методы действий и их параметры

Методы действий (action methods) представляют такие методы контроллера, которые обрабатывают запросы по определенному **URL**. Например, возьмем проект из предыдущей главы. В нем был определен следующий контроллер:

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

namespace BookStore.Controllers
{
    public class HomeController : Controller
    {
        // создаем контекст данных
        BookContext db = new BookContext();

        public ActionResult Index()
        {
           IEnumerable<Book> books = db.Books;
           ViewBag.Books = books;
           return View();
        }

        [HttpGet]
        public ActionResult Buy(int id)
        {
            ViewBag.BookId = id;
            return View();
        }
        [HttpPost]
        public string Buy(Purchase purchase)
        {
            purchase.Date = DateTime.Now;
```

```
            db.Purchases.Add(purchase);
            db.SaveChanges();
            return "Спасибо," + purchase.Person + ", за покупку!";
        }
    }
}
```

Здесь методы **Index** и **Buy** являются методами действий или просто действиями контроллера. При получении запроса типа **/Home/Index** контроллер передает обработку запроса действию **Index**.

Так как запросы бывают разных типов, например, **GET** и **POST**, фреймворк **ASP.NET MVC** позволяет определить тип обрабатываемого запроса для действия, применив к нему соответствующий атрибут: **[HttpGet], [HttpPost], [HttpDelete]** или **[HttpPut]**. Так, действие **Buy** разбито на два метода, по одному для каждого типа запроса.

Однако не все методы контроллера являются методами действий. Методы действий всегда имеют модификатор **public**. Закрытых приватных методов действий не бывает. Но контроллер может также включать и обычные методы, которые могут использоваться в вспомогательных целях. Например,

```
[HttpPost]
        public string Buy(Purchase purchase)
        {
            purchase.Date = getToday();
            db.Purchases.Add(purchase);
            db.SaveChanges();
            return "Спасибо, " + purchase.Person + ", за покупку!";
        }

private DateTime getToday()
        {
            return DateTime.Now;
        }
```

Соответственно мы не можем отправить из браузера запрос **Home/getToday/**, потому что метод **getToday** не является методом действия.

Передача данных в контроллеры и параметры

В приложении из предыдущей главы метод **Buy** использовал параметр **purchase**. Так как данный метод обрабатывает **POST**-запросы, то мы можем отправить ему следующую форму:

```
<form method="post" action="">
        <input type="hidden" value="@ViewBag.BookId" name="BookId" />
        <p>Введите свое имя </p>
        <input type="text" name="Person" />
        <p>Введите адрес :</p>
        <input type="text" name="Address" />
        <input type="submit" value="Отправить" />
</form>
```

Значение атрибута **name** у всех полей на этой форме соответствует названию свойства модели, поэтому система автоматически свяжет значения полей с соответствующими свойствами. А в методе **Buy** весь этот набор свойств превратится в модель **Purchase**.

Кроме **POST**-запросов у нас есть также **GET**-запросы, при которых все параметры передаются в строке запроса. Например, вторая версия метода **Buy** в качестве параметра принимает значение типа **int:**

```
public ActionResult Buy(int id)
```

Стандартный **GET**-запрос принимает примерно следующую форму: **название_ресурса?параметр1=значение1&параметр2=значение2**.

То есть запрос к данному методу мог бы выглядеть так: **Home/Buy?id=2**. Название параметров метода должно совпадать с названием параметров в строке запроса. Благодаря этому система сможет их автоматически связать. А в самом методе мы сможем получить этот параметр и использовать его по своему усмотрению.

Кроме того, система маршрутизации позволяет создавать маршруты. Например, по умолчанию в проекте **MVC** определяется следующий маршрут: **Контроллер/Метод/id**.

Последний параметр является опциональным. И благодаря этому мы можем передать параметр **id** и так: **Home/Buy/2**

Для примера определим действие, которое будет подсчитывать площадь треугольника:

```
public string Square(int a, int h)
        {
            double s = a * h / 2;
            return "<h2>Площадь треугольника с основанием " + a +
                    " и высотой " + h + " равна " + s + "</h2>";
        }
```

В этом случае мы можем обратиться к действию, набрав в адресной строке **Home/Square?a=10&h=3**, и приложение выдало бы нам нужный результат.

Мы также можем задать для параметров значения по умолчанию:

```
public string Square(int a=10, int h=3)
        {
            double s = a * h / 2;
            return "<h2>Площадь треугольника с основанием " + a +
                    " и высотой " + h + " равна " + s + "</h2>";
        }
```

В этом случае при запросе страницы мы можем указать только один параметр или вообще не указывать (**Home/Square?h=5**).

Получение данных из контекста запроса

Кроме того, мы можем получить параметры, да и не только параметры, но и другие данные, связанные с запросом, из объектов контекста запроса. Нам доступны следующие объекты контекста: **Request**, **Response**, **RoutedData**, **HttpContext** и **Server**.

Объект **Request** содержит коллекцию **Params**, которая хранит все параметры, переданные в запросы. И мы их можем получить:

```
public string Square()
        {
            int a = Int32.Parse(Request.Params["a"]);
            int h = Int32.Parse(Request.Params["h"]);
            double s = a * h / 2;
            return "<h2>Площадь треугольника с основанием " + a + " и
высотой " + h + " равна " + s + "</h2>";
        }
```

Результаты действий

Когда пользователь обращается к ресурсу, как правило, он ожидает получить определенный ответ, например, в виде веб-страницы с некоторыми данными. На стороне сервера метод контроллера, получая параметры, обрабатывает их и формирует некоторый ответ в виде результата действия.

В прошлой теме в примере с вычислением площади треугольника мы возвращали html-код в виде строки. Но, как правило, возвращаемым результатом является объект класса, производного от **ActionResult**.

ActionResult представляет собой абстрактный класс, в котором определен один метод **ExecuteResult**, переопределяемый в классах-наследниках:

```
public abstract class ActionResult
    {
        public abstract void ExecuteResult(ControllerContext context);
    }
```

Создадим свои результаты действий. Они будут очень простыми. Возьмем какой-нибудь проект, например, из прошлой главы, и добавим в него новую папку **Util**, которая будет содержать новый классы. После добавления папки добавим в нее первый класс. Назовем его **HtmlResult**. Он у нас будет содержать следующий код:

```
using System.Web.Mvc;

namespace BookStore.Util
{
    public class HtmlResult : ActionResult
    {
        private string htmlCode;
        public HtmlResult(string html)
        {
            htmlCode = html;
        }
        public override void ExecuteResult(ControllerContext context)
        {
            string fullHtmlCode = "<!DOCTYPE html><html><head>";
            fullHtmlCode += "<title>Главная страница</title>";
            fullHtmlCode += "<meta charset=utf-8 />";
            fullHtmlCode += "</head> <body>";
            fullHtmlCode += htmlCode;
            fullHtmlCode += "</body></html>";
            context.HttpContext.Response.Write(fullHtmlCode);
        }
    }
}
```

В конструкторе класса **HtmlResult** получаем переданный **html**-код, а в методе **Execute** вставляем его в общее окружение, чтобы получилась полноценная **html**-страница, и пишем ее в выходной поток: **context.HttpContext.Response.Write(fullHtmlCode)**;

Чтобы использовать этот класс подключим в контроллер пространство имен нового класса: **using BookStore.Util;** и добавим новый метод:

```
public ActionResult GetHtml()
        {
            return new HtmlResult("<h2>Привет мир!</h2>");
        }
```

И обратившись к этому методу из браузера, например, **Home/GetHtml**, мы получим **html**-страничку. Хотя данный пример довольно примитивен, но в целом он демонстрирует, как работают классы результатов действий.

Создадим еще один класс результатов. Добавим в папку **Util** новый класс **ImageResult**:

```
using System.Web.Mvc;

namespace BookStore.Util
{
    public class ImageResult : ActionResult
    {
        private string path;
        public ImageResult(string path)
        {
            this.path = path;
        }
        public override void ExecuteResult(ControllerContext context)
        {
            context.HttpContext.Response.Write("<div
style='width:100%;text-align:center;'>" +
                "<img style='max-width:600px;' src='" + path + "'
/></div>");
        }
    }
}
```

Данный класс не сложнее предыдущего и просто отдает изображение к **html**-коде. Тогда метод, использующий данный результат действий, мог бы выглядеть так:

```
public ActionResult GetImage()
        {
            string path = "../Images/visualstudio.png";
            return new ImageResult(path);
        }
```

Здесь предполагается, что в проекте есть папка **Images**, в которой имеется изображение **visualstudio.png**. И тогда, если мы в браузере обратимся к этому действию, например, **Home/GetImage**, то сможем увидеть изображение.

Встроенные классы, производные от ActionResult

В реальности нам вряд ли потребуется часто создавать свои классы для обработки результата действия. Фреймворк **ASP.NET MVC** предлагает нам богатую палитру классов результатов действий, которые охватывают большинство, если не все возможные ситуации.

- **ContentResult**: пишет указанный контент напрямую в ответ в виде строки, практически как предыдущие примеры.

Так, следующий пример:

```
public string Square(int a, int h)
        {
            int s = a * h / 2;
            return "<h2>Площадь треугольника с основанием " + a +
                    " и высотой " + h + " равна " + s + "</h2>";
        }
```

Можно переписать с использованием **ContentResult** следующим образом:

```
public ContentResult Square(int a, int h)
        {
            int s = a * h / 2;
            return Content("<h2>Площадь треугольника с основанием " + a +
                    " и высотой " + h + " равна " + s + "</h2>");
        }
```

Даже если мы оставим в качестве возвращаемого результата тип **string**, то фреймворк увидит, что возвращаемый тип не является объектом **ActionResult**. И тогда автоматически создается объект **ContentResult** для возвращаемой строки.

- **EmptyResult**: по сути ничего не делает, отправляет пустой ответ
- **FileResult**: является базовым классом для всех объектов, пишущих бинарный ответ в выходной поток. Предназначен для отправки
- **FileContentResult**: класс, производный от **FileResult**, пишет в ответ массив байтов
- **FilePathResult**: также производный от **FileResult** класс, пишет в ответ файл, находящийся по заданному пути
- **FileStreamResult**: класс, производный от **FileResult**, пишет бинарный поток в выходной ответ

- **HttpStatusCodeResult**: результат действия, который возвращает клиенту определенный статусный код **HTTP**
- **HttpUnauthorizedResult**: класс, производный от **HttpStatusCodeResult**. Возвращает клиенту ответ в виде статусного кода **HTTP 401**, указывая, что пользователь не прошел авторизацию и не имеет прав доступа к запрошенному ресурсу.
- **HttpNotFoundResult**: производный от **HttpStatusCodeResult**. Возвращает клиенту ответ в виде статусного кода **HTTP 404**, указывая, что запрошенный ресурс не найден.
- **JavaScriptResult**: возвращает в ответ в качестве содержимого код **JavaScript**
- **JsonResult**: возвращает в качестве ответа объект или набор объектов в формате **JSON**
- **PartialViewResult**: производит рендеринг частичного представления в выходной поток
- **RedirectResult**: перенаправляет пользователя по другому адресу **URL**, возвращая статусный код **302** для временной переадресации или код **301** для постоянной переадресации зависимости от того, установлен ли флаг Permanent.
- **RedirectToRouteResult**: класс работает подобно **RedirectResult**, но перенаправляет пользователя по определенному адресу **URL**, указанному через параметры маршрута
- **ViewResult**: производит рендеринг представления и отправляет результаты рендеринга в виде **html**-страницы клиенту

Рассмотрим подробнее работу некоторых из этих классов.

ViewResult и генерация представлений

Класс **ViewResult** является наиболее часто возвращаемым результатом действий контроллера. Он производит рендеринг представления в веб-страницу и возвращает ее в виде ответа клиенту.
Чтобы возвратить объект **ViewResult** используется метод **View**:

```
public class HomeController : Controller
    {
       public ActionResult Index()
        {
           return View();
        }
     }
```

Вызов метода **View** возвращает объект **ViewResult**. Затем уже **ViewResult** производит рендеринг определенного представления в ответ. По умолчанию контроллер производит поиск представления в проекте по следующему пути: **/Views/Имя_контроллера/Имя_представления.cshtml**

Согласно настройкам по умолчанию, если представление не указано явным образом, то в качестве представления будет использоваться то, имя которого совпадает с именем действия контроллера. Например, вышеопределенное действие **Index** по умолчанию будет производить поиск представления **Index.cshtml** в папке **/Views/Home/**.
Однако можно также задать имя представления явным образом:

```
public class HomeController : Controller
    {
       public ActionResult Index()
        {
           return View("Index");
        }
     }
```

В итоге в качестве представления будет выбрано представление **/Views/Home/Index.cshtml**. Мы также можем полностью переопределить путь, по которому система будет искать представление:

```
public class HomeController : Controller
    {
       public ActionResult Index()
        {
           return View("~/Views/Some/Index.cshtml");
        }
```

```
    }
```

Правда, если такого пути не окажется, то приложение выбросит ошибку.

Передача данных из контроллера в представление

Существуют различные способы передачи данных из контроллера в представление. Один из них представляет использование объекта **ViewData**. **ViewData** представляет словарь из пар ключ-значение:

```
public ViewResult SomeMethod()
        {
            ViewData["Head"] = "Привет мир!";
            return View("SomeView");
        }
```

В этом случае в представлении **SomeView.cshtml** можно получить передаваемую строку следующим образом:

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>SomeView</title>
</head>
<body>
    <div>
        <h2>@ViewData["Head"]</h2>
    </div>
</body>
</html>
```

Еще один способ передачи данных представляет объект **ViewBag**. Этот объект позволяет определить различные свойства и присвоить им любое значение. Так, мы могли бы переписать предыдущий пример следующим образом:

```
public ViewResult SomeMethod()
        {
            ViewBag.Head = "Привет мир!";
```

```
            return View("SomeView");
        }
```

И таким же образом изменить представление:

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>SomeView</title>
</head>
<body>
    <div>
        <h2>@ViewBag.Head</h2>
    </div>
</body>

</html>
```

И не важно, что изначально объект **ViewBag** не содержит никакого свойства **Head**, оно определяется динамически. При этом свойства **ViewBag** могут содержать не только простые объекты типа **string** или **int**, но и сложные данные. Так, в примере из прошлой главы мы добавляли в объект **ViewBag** список объектов модели:

```
public ActionResult Index()
        {
            IEnumerable<Book> books = db.Books;
            ViewBag.Books = books;
            return View();
        }
```

Хотя **ViewData** и **ViewBag** и похожи, в то же время они не полностью эквивалентны. Так, например, нельзя передавать динамические значения из **ViewBag** в методы расширения в представлениях.

Например, мы не можем написать **@Html.TextBox("name", ViewBag.Name)**, так как компилятор **C#** должен знать тип каждого параметра во время компиляции, чтобы выбрать нужный метод расширения. В этом случае нам надо либо использовать **ViewData:@Html.TextBox("name",**

ViewData["Name"]), либо применить приведение типов: **@Html.TextBox("name", (string)ViewBag.Name).**

И еще один способ предлагает объект **TempData**. **TempData** представляет словарь, хранящий пары ключ-значение, как и **ViewData**, но его использование немного отличается. **TempData** позволяет сохранять переданное значение в течении всего текущего запроса. Использование **TempData** аналогично работе с **ViewData**.

Переадресация и отправка кодов статуса и ошибок

Существует два вида переадресации: временная и постоянная. И в зависимости от вида переадресации при ее выполнении сервер посылает браузерам один из двух кодов **HTTP**:

- статусный код **301** представляет постоянную переадресацию. При данном типе переадресации предполагается, что запрашиваемый документ окончательно перемещен в другое место. После получения данного статусного кода браузер может автоматически настраивать запросы на новый ресурс, даже если старый ресурс со временем перестанет применять переадресацию. Поэтому данный способ использовать нежелательно.
- статусный код **302** представляет временную переадресацию. При временной переадресации считается, что запрашиваемый документ временно перемещен на другую страницу.

В обоих случаях для переадресации будет использоваться объект **RedirectResult**, однако метод, возвращающий данный объект, будет отличаться.

Для временной переадресации применяется метод **Redirect**:

```
public RedirectResult SomeMethod()
        {
            return Redirect("/Home/Index");
        }
```

Для постоянной переадресации подобным образом используется метод **RedirectPermanent**:

```
public RedirectResult SomeMethod()
        {
            return RedirectPermanent("/Home/Index");
        }
```

При этом нам необязательно возвращать из метода объект **RedirectResult**. Нередко возникает ситуация, когда в зависимости от некоторых условий требуется направить пользователя по одному адресу, либо переадресовать на другой ресурс. Типичная ситуация: авторизация пользователя - если он авторизован, то ему отображается требуемая веб-страница, а если нет, то он перенаправляется на страницу для логина. Например:

```
public ActionResult Buy(int id)
        {
            if (id > 3)
            {
                return Redirect("/Home/Index");
            }
            ViewBag.BookId = id;
            return View();
        }
```

Если в качестве параметра будет передано число больше **3**, то произойдет редирект на **Home/Index**. В остальных случаях пользователю будет возвращаться представление.

Еще один класс для создания переадресации - **RedirectToRouteResult** - позволяет выполнить более детальную настройку перенаправлений. Он возвращается двумя методами: **RedirectToAction** и **RedirectToRoute**.

Метод **RedirectToRoute** позволяет произвести перенаправление по определенному маршруту внутри домена:

```
public RedirectToRouteResult SomeMethod()
        {
            return RedirectToRoute(new { controller = "Home", action =
"Index" });
        }
```

Метод **RedirectToAction** позволяет перейти к определенному действию определенного контроллера. Он также позволяет задать передаваемые параметры:

```
public RedirectToRouteResult SomeMethod()
        {
            return RedirectToAction("Square", "Home", new { a = 10, h =
12 });
        }
```

Методы **Redirect/RedirectToAction** представлют временную переадресацию. Но они имеют свои двойники для создания постоянной переадресации: **RedirectPermanent/RedirectToActionPermanent**. Их действие аналогично, разница лишь в том, что они отправляют браузеру статусный код **301**. Однако методы **RedirectPermanent** и **RedirectToActionPermanent** не рекомендуется использовать, а если и использовать, то с осторожностью. Так как неправильно настроенная постоянная переадресация может ухудшить позиции в поисковиках или способствовать полному выпадению сайта из поиска.

Отправка ошибок и статусных кодов

Иногда возникает необходимость отправить сообщения об ошибках при доступе к тому или иному ресурсу. Обычно, если ресурс недоступен, **mvc**-фреймворк автоматически отреагирует на эту ситуацию, отправив соответствующий статусный код.

Но в некоторых ситуациях нам нужно более тонко реагировать на полученный запрос. Например, у нас есть контент, к которому установлены возрастные ограничения. Мы смотрим введенный возраст, и если он попадает под ограничение, мы можем выслать статусный код ошибки:

```
public ActionResult Check(int age)
        {
            if (age < 21)
            {
                return new HttpStatusCodeResult(404);
            }
            return View();
        }
```

Подобным образом мы можем послать браузеру любой другой статусный код.

В качестве альтернативы также можно возвращать объект **HttpNotFoundResult** с помощью метода **HttpNotFound**

```
public ActionResult CheckAge(int age)
        {
            if (age < 21)
            {
                return HttpNotFound();
            }
            return View();
        }
```

И еще один класс, предназначенный для отправки статусных кодов - класс **HttpUnauthorizedResult**. Он извещает пользователя, что тот не имеет права доступа к ресурсу, отправляя браузеру статусный код **401**:

```
public ActionResult AgeCheck(int age)
        {
            if (age < 21)
            {
                return new HttpUnauthorizedResult();
            }
            return View();
        }
```

Отправка файлов в ASP.NET MVC 5

Для отправки клиенту файлов предназначен **FileResult**. Однако так как это абстрактный класс, то фактически мы будем иметь дело с его наследниками:

- **FileContentResult**: отправляет клиенту массив байтов, считанный из файла
- **FilePathResult**: представляет простую отправку файла напрямую с сервера
- **FileStreamResult**: данный класс создает поток - объект System.IO.Stream, с помощью которого считывает и отправляет файл клиенту

Во всех трех случаях для отправки файлов применяется метод **File**, который и возвращает объект **FileResult**. Только в зависимости от выбранного способа используется соответствующая перегруженная версия этого метода.

Чтобы отправить файл из файловой системы (то есть использование объекта **FilePathResult**), нам надо указать в методе **File** три параметра: **путь к файлу на стороне сервера**, **тип содержимого** и **имя файла** для принимающей стороны (имя файла необязательно, и можно обойтись в принципе только двумя параметрами).

```
public FileResult GetFile()
        {
            // Путь к файлу
            string file_path = Server.MapPath("~/Files/PDFIcon.pdf");
            // Тип файла - content-type
            string file_type = "application/pdf";
            // Имя файла - необязательно
            string file_name = "PDFIcon.pdf";
            return File(file_path, file_type, file_name);
        }
```

Предполагается, что у нас в проекте есть папка **Files**, в которой лежит файл **PDFIcon.pdf**. Метод **Server.MapPath** позволяет построить полный путь к ресурсу из каталога в проекте. Но также можно использовать и абсолютные пути, обращаясь к любому файлу в файловой системе, например,

string file_path = @"C:\Book\PDFIcon.pdf";

И, при обращении, например, по пути **Home/GetFile** нам будет предложено сохранить данный файл на локальном компьютере.

Похожим образом работает и классы **FileContentResult**, только вместо имени файла в методе **File** указывается массив байтов, в который был считан файл:

```
// Отправка массива байтов
public FileResult GetBytes()
        {
            string path = Server.MapPath("~/Files/PDFIcon.pdf");
            byte[] mas = System.IO.File.ReadAllBytes(path);
            string file_type = "application/pdf";
            string file_name = "PDFIcon.pdf";
            return File(mas, file_type, file_name);
        }
```

И если мы хотим возвратить объект **FileStreamResult**, то в качестве первого аргумента в методе **File** идет объект **Stream** для отправляемого файла:

```
// Отправка потока
public FileResult GetStream()
        {
            string path = Server.MapPath("~/Files/PDFIcon.pdf");
            // Объект Stream
            FileStream fs = new FileStream(path, FileMode.Open);
            string file_type = "application/pdf";
            string file_name = "PDFIcon.pdf";
            return File(fs, file_type, file_name);
        }
```

Контекст запроса HttpContext. Куки. Сессии

Нередко для обработки запроса требуется информация о контексте запроса: какой у пользователя браузер, **IP**-адрес, с какой страницы или сайта пользователь попал к нам. И **ASP.NET MVC** позволяет получить всю эту информацию, используя объект **HttpContext**.

Хотя в контроллере мы также можем обратиться к объекту **ControllerContext**, который имеет свойство **HttpContext** и по сути предоставляет доступ к той же функциональности и информации.

Но в то же время между ними есть некоторые различия. Объект **HttpContext** описывает данные конкретного **http-запроса**, который обрабатывается приложением. А **ControllerContext** описывает данные **http-запроса** непосредственно по отношению к данному контроллеру.

Вся информация о контексте запроса доступна нам через свойства объекта **HttpContext**. Так, все данные запроса содержится в свойстве **Request.HttpContext.Request** представляет объект класса, унаследованного от **HttpRequestBase**, и поэтому содержит все его свойства. Рассмотрим некоторые из них:

- Получение браузера пользователя: **HttpContext.Request.Browser**
- Иногда просто браузера недостаточно, тогда можно обратиться к агенту пользователя: **HttpContext.Request.UserAgent**
- Получение url запроса: **HttpContext.Request.RawUrl**
- Получение IP-адреса пользователя: **HttpContext.Request.UserHostAddress**
- Получение реферера: HttpContext.Request.UrlReferrer == null ? "" : HttpContext.Request.UrlReferrer.AbsoluteUri Так как реферер может быть не определен, то сначала смотрим, не равен ли он null

Например: (http://localhost:2044/Home/MyIndex)

```
public string MyIndex()
        {
            string browser = HttpContext.Request.Browser.Browser;
            string user_agent = HttpContext.Request.UserAgent;
            string url = HttpContext.Request.RawUrl;
            string ip = HttpContext.Request.UserHostAddress;
            string referrer = HttpContext.Request.UrlReferrer == null ?
"" : HttpContext.Request.UrlReferrer.AbsoluteUri;
            return "<p>Browser: " + browser + "</p><p>User-Agent: " +
user_agent + "</p><p>Url запроса: " + url +
                "</p><p>Реферер: " + referrer + "</p><p>IP-адрес: " + ip
+ "</p>";
        }
```

Отправка ответа

Если **HttpContext.Request** содержит информацию о запросе, то свойство **HttpContext.Response** управляет ответом. Оно представляет объект **HttpResponse**, который передает на сторону клиента некоторые значения: **куки**, **служебные заголовки**, **сам ответ** в виде кода **html**.

Например, установим кодировку ответа: **HttpContext.Response.Charset = "iso-8859-2";**

Методы объекта **HttpResponse** позволяют управлять ответом. Например, метод **AddHeader** позволяет добавить к ответу дополнительный заголовок.

Кроме того, нам необязательно вызывать метод **View()** в действия контроллера, чтобы отправить клиенту ответ запроса. Мы вполне можем воспользоваться методом **HttpContext.Response.Write()**:

(http://localhost:2044/Home/ContextData)

```
public string ContextData()
        {
            HttpContext.Response.Write("<h1>Hello World</h1>");

            string user_agent = HttpContext.Request.UserAgent;
            string url = HttpContext.Request.RawUrl;
            string ip = HttpContext.Request.UserHostAddress;
            string referrer = HttpContext.Request.UrlReferrer == null ?
"" : HttpContext.Request.UrlReferrer.AbsoluteUri;
            return "<p>User-Agent: " + user_agent + "</p><p>Url запроса:
" + url +
                "</p><p>Реферер: " + referrer + "</p><p>IP-адрес: " + ip
+ "</p>";
        }
```

Либо так:

```
public void ContextData()
        {
            HttpContext.Response.Write("<h1>Hello World</h1>");
        }
```

Метод **HttpContext.Response.Write** здесь добавляет в поток определенное содержимое, переданное в качестве параметра. Но в реальности, конечно, проще использовать методы, генерирующие объекты **ActionResult**, например, представления.

Определение пользователя

Также объект **HttpContext** содержит информацию о пользователе в свойстве **HttpContext.User**:

```
bool IsAdmin = HttpContext.User.IsInRole("admin");
// определяем, принадлежит ли пользователь к администраторам
bool IsAuth = HttpContext.User.Identity.IsAuthenticated;
// аутентифицирован ли пользователь
string login = HttpContext.User.Identity.Name;
// логин авторизованного пользователя
```

Подробнее об идентификации пользователя можно узнать в главе про аутентификацию и авторизацию.

Работа с куки

Чтобы получить куки, нам надо воспользоваться свойством **HttpContext.Request.Cookies**:

```
public void GetCookie()
        {
            string id = HttpContext.Request.Cookies["id"].Value;
        }
```

В данном случае, если у нас установлена на стороне клиента куки "id", то мы получим ее значение.

Однако прежде чем получать значения куки, их естественно надо установить. Для установки значения куки мы можем использовать свойство **HttpContext.Response**. Например, установим в куки значение "id":
HttpContext.Response.Cookies["id"].Value = "ca-4353w";

Сессии

Сессии также, как и куки, используются для хранения некоторых данных, которые можно получить в любом месте приложения. Для работы с ними используется объект **Session**. Например, установим в одном методе контроллера мы можем установить значение сессии:

```
public ActionResult MyIndex()
        {
            Session["name"] = "Tom";
            return View();
        }
```

А получить можно в другом методе:

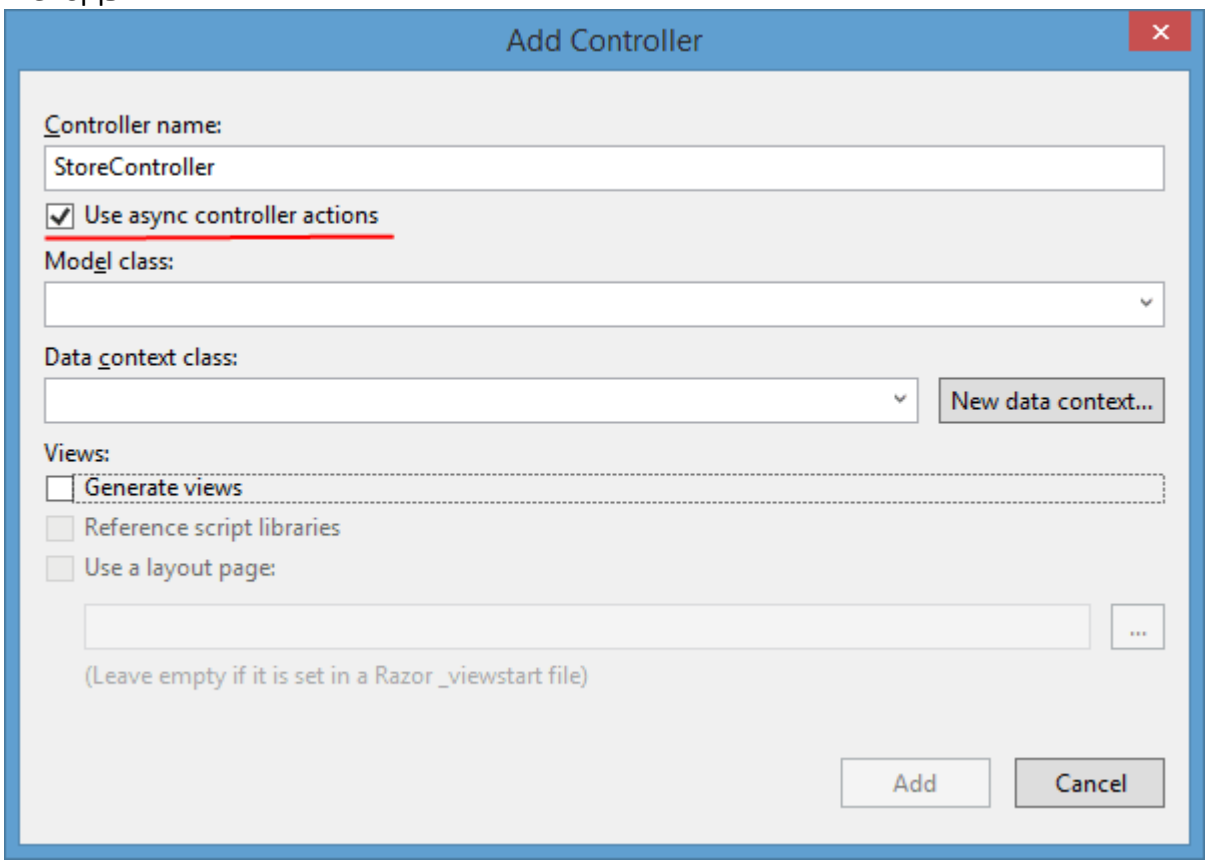
```
public string GetName()
        {
            var val = Session["name"];
            return val.ToString();
        }
```

Если мы хотим удалить значение сессии для ключа name, мы можем просто присвоить значение null: **Session["name"]=null;**

Асинхронные методы

Одним из ключевых нововведений последних версий фреймворка **.NET** стала асинхронность. Хотя фреймворк и раньше позволял использовать асинхронные методы, но с появлением библиотеки **Task Parallel Library** работа с асинхронным кодом была предельно упрощена, а сам формат работы изменился. Были добавлены новые возможности по созданию асинхронных методов с использованием новых ключевых слов, таких как **async** и **await**.

При создании нового контроллера мы в настройках уже можем указать, как нам нужен контроллер - синхронный или асинхронный. По умолчанию **Visual Studio** добавляет в проект стандартные контроллеры, методы которых, как правило, возвращают объект **ActionResult**. Но если мы при добавлении контроллера в папку **Controllers** выберем тип **MVC 5 Controller with views, using Entity Framework**, то в окне настройки нового контроллера специальное поле позволит нам указать, что новый контроллер будет содержать асинхронные методы:



Для чего нужны вообще асинхронные методы в контроллерах? Асинхронные методы позволяют оптимизировать производительность приложения и предназначены прежде всего для обработки таких запросов, которые занимают или могут занять довольно продолжительное время,

например, обращение к базе данных или обращение к внешнему сетевому ресурсу для получения большой порции данных. Применение асинхронных методов позволяет приложению параллельно с выполнением асинхронного кода выполнять также другие запросы.

Чтобы понять различие между синхронными и асинхронными методами, рассмотрим, как **IIS** обрабатывает входящие запросы. Веб-сервер поддерживает пул потоков, которые обслуживают запросы. При обращении пользователя к веб-ресурсу **IIS** выделяет поток из пула для обслуживания данного запроса. И пока данный поток не обработает предназначенный для него запрос, другие запросы он обрабатывать не может.

Однако предположим, что метод контроллера в процессе обработки запроса должен выполнить запрос к другому ресурсу или к базе данных. Запрос к сетевому ресурсу или **БД** сам по себе может занять некоторое время. При синхронной обработке поток, обрабатывающий запрос, временно блокируется, пока сетевой ресурс или **БД** не возвратят нужные нам данные.

И если обработка запроса блокируется очень долго, то **IIS** начинает задействовать для обслуживания других входящих запросов новые потоки. Однако есть ограничения на общее количество потоков. Когда количество потоков достигает предела, то вновь входящие запросы помещаются в очередь ожидания. Однако и тут есть ограничение на количество запросов в очереди. И когда это количество превышает предел, то **IIS** просто отклоняет все остальные запросы с помощью статусного кода **503 (Service Unavailable)**.

При асинхронной обработке поток не ждет, пока **БД** вернет ему данные, а начинает обрабатывать запрос от другого пользователя. Но когда, наконец, с сетевого ресурса или БД придут нужные данные, поток возвращается к обработке ранее обрабатываемого запроса в обычном режиме.

Перейдем непосредственно к коду. Для создания асинхронных методов используются модификаторы **async** и **await**, которые позволяют выполнять продолжительные операции без блокирования основного потока.
Сравним на примере вызов синхронного и асинхронного метода:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using BookStore.Models;
using System.Threading.Tasks;
using System.Data.Entity;


namespace BookStore.Controllers
```

```
{

    public class HomeController : Controller
    {
        // маълумотлар контекстини ҳосил қиламиз
        BookContext db = new BookContext();

        public ActionResult Index()
        {
            IEnumerable<Book> books = db.Books;
            ViewBag.Books = books;
            return View();
        }

        // асинхронный метод
        public async Task<ActionResult> BookList()
        {
            IEnumerable<Book> books = await Task.Run(() => db.Books);
            ViewBag.Books = books;
            return View("Index");
        }


    }
}
```

Оба метода выполняют одну и ту же операцию - извлечение данных из БД и получают идентичные результаты. Но если первый синхронный метод **Index** представляет привычную для нас запись, то асинхронный метод **BookList** уже выглядит необычно.

Этот метод возвращает не объект **ActionResult**, а объект **Task<ActionResult>**. **Task** представляет асинхронную операцию, выполняющуюся продолжительное время.

Кроме того, чтобы обозначить метод как асинхронный, перед возвращаемым типом ставится ключевое слово **async**.

Третьим ключевым моментом является использование ключевого слова **await**. Оно применяется в асинхронных методах, чтобы приостановить выполнение этого метода до тех пор, пока ожидаемая задача не завершится. В нашем случае такой задачей является получение данных из БД.

Но также следует учитывать, что **await** используется с методами, возвращающими объект **`Task`**. Поэтому получение данных их БД мы обернули в такое выражение `await Task.Run(() => db.Books;`.

Метод **`Task.Run`** в качестве параметра получает некоторое действие в виде лямбда-выражения. А этим действием и является извлечение данных из **БД: db.Books.**

Но добавление асинхронности произошло по всему фронту технологий **.NET**. В частности, в новую версию **Entity Framework 6** также была добавлена асинхронность на уровне отдельных методов, которые мы можем применять при операциях над данными. Например, предыдущий пример асинхронного метода **BookList** мы можем переписать следующим образом:

```
public async Task<ActionResult> BookList()
        {
            ViewBag.Books = await db.Books.ToListAsync();
            return View("Index");
        }
```

Метод **db.Books.ToListAsync()** возвращает список объектов. **Entity Framework 6** предлагает еще ряд асинхронных методов для работы с **БД**, с которыми мы позже познакомимся.

Когда следует использовать асинхронные методы? В первую очередь их предпочтительно использовать при запросах к **БД**, к внешнем сетевым ресурсам, однако в конечном счете, что лучше синхронность или асинхронность решает уже сам разработчик исходя из конкретной задачи.

Представления

Введение в представления

Хотя работа приложения **MVC** управляется главным образом контроллерами, но непосредственно пользователю приложение доступно в виде представления, которое и формирует внешний вид приложения. В **ASP.NET MVC 5** представления - это файлы с расширением **cshtml**, которые содержат код пользовательского интерфейса в основном на языке **html**. Стандартное представление:

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>SomeView</title>
</head>
<body>
    <div>
        <h2>@ViewBag.Message</h2>
    </div>
</body>
</html>
```

Хотя представление содержит главным образом, код **html**, оно не является **html**-страницей. При компиляции приложения на основе требуемого представления сначала генерируется класс на языке **C#**, а затем этот класс компилируется. Так, из выше приведенного представления будет генерироваться примерно в такой класс:

```
#pragma checksum "c:\users\hp\documents\visual studio
2013\Projects\MVC\BookStore\BookStore\Views\Home\SomeView.cshtml" "{ff1816ec-aa5e-4d10-87f7-
6f4963833460}" "1F05F4D370C9D00F8CBDFB8BD1F51D74189D0617"

namespace ASP {
    using System;
    using System.Collections.Generic;
    using System.IO;
    using System.Linq;
    using System.Net;
    using System.Web;
    using System.Web.Helpers;
```

```
    using System.Web.Security;
    using System.Web.UI;
    using System.Web.WebPages;
    using System.Web.Mvc;
    using System.Web.Mvc.Ajax;
    using System.Web.Mvc.Html;
    using System.Web.Optimization;
    using System.Web.Routing;
    using BookStore;

    public class _Page_Views_Home_SomeView_cshtml : System.Web.Mvc.WebViewPage<dynamic>
    {

    #line hidden

    public _Page_Views_Home_SomeView_cshtml() {
    }

    protected ASP.global_asax ApplicationInstance {
    get {
    return ((ASP.global_asax)(Context.ApplicationInstance));
    }
    }

    public override void Execute() {
    BeginContext("~/Views/Home/SomeView.cshtml", 0, 2, true);

    WriteLiteral("\r\n");

    EndContext("~/Views/Home/SomeView.cshtml", 0, 2, true);


    #line 2 "c:\users\hp\documents\visual studio
2013\Projects\MVC\BookStore\BookStore\Views\Home\SomeView.cshtml"

    Layout = null;


    #line default
    #line hidden
    BeginContext("~/Views/Home/SomeView.cshtml", 27, 48, true);

    WriteLiteral("\r\n\r\n<!DOCTYPE HTML>
    \r\n\r\n
    <html>
    \r\n
    <head>
        \r\n
        <meta"); endcontext("~ />Views/Home/SomeView.cshtml", 27, 48, true);

        BeginContext("~/Views/Home/SomeView.cshtml", 75, 16, true);

        WriteLiteral(" name=\"viewport\"");

        EndContext("~/Views/Home/SomeView.cshtml", 75, 16, true);

        BeginContext("~/Views/Home/SomeView.cshtml", 91, 29, true);

        WriteLiteral(" content=\"width=device-width\"");

        EndContext("~/Views/Home/SomeView.cshtml", 91, 29, true);
```

```
        BeginContext("~/Views/Home/SomeView.cshtml", 120, 74, true);

        WriteLiteral(" />\r\n
        <title>SomeView</title>\r\n
    </head>\r\n
    <body>
        \r\n    <div>
            \r\n       <h2>
                ");

                EndContext("~/Views/Home/SomeView.cshtml", 120, 74, true);

                BeginContext("~/Views/Home/SomeView.cshtml", 195, 15, false);


                #line 15 "c:\users\hp\documents\visual studio
2013\Projects\MVC\BookStore\BookStore\Views\Home\SomeView.cshtml"
                Write(ViewBag.Message);


                #line default
                #line hidden
                EndContext("~/Views/Home/SomeView.cshtml", 195, 15, false);

                BeginContext("~/Views/Home/SomeView.cshtml", 210, 38, true);

                WriteLiteral("
            </h2> \r\n
        </div>\r\n
    </body>\r\n
</html>\r\n");

EndContext("~/Views/Home/SomeView.cshtml", 210, 38, true);

        }
    }
}
```

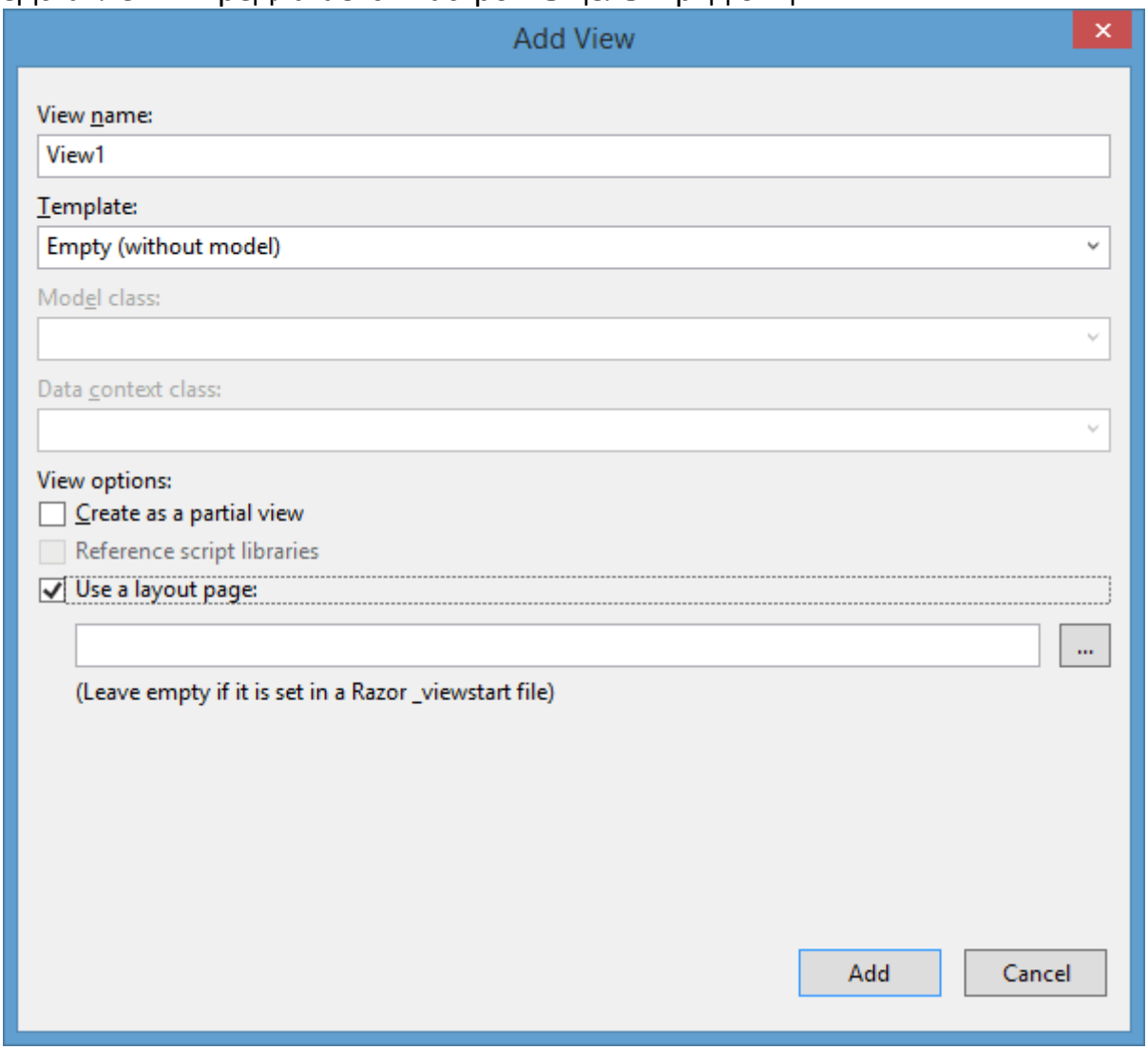
Код, конечно, не самый читабельный, особенно если не знать, что делают все эти классы и методы, но здесь мы можем увидеть, что при компиляции создается класс, наследующий от класса **System.Web.Mvc.WebViewPage<T>**, где **T** - это класс модели, которая будет использоваться.

Но так как представление не строго типизированное, поэтому вместо имени класса модели идет ключевое слово **dynamic**. Все действия данного класса заключены в методе **Execute**, в котором с помощью метода **WriteLiteral** обрабатываются все имеющиеся в представлении элементы разметки **html**.

Найти сгенерированные из представлений файлы кода можно по пути **C:\Users\Имя_Логина\AppData\Local\Temp\Temporary ASP.NET Files\root.** Правда, все папки имеют зашифрованные имена, поэтому чтобы определить нужную папку приложения, нужно будет затратить некоторое время на поиск. Кроме того, сами генерируемые файлы кода также имеют зашифрованные имена. Например, выше приведенный класс в моем случае имеет имя **App_Web_nri53fza.1.cs** и находится в папке **root\307f1c1d\a36bbd4f**.

Создание нового представления

Для создания нового представления выберем в проекте папку **Views** и в нем нажмем правой кнопкой на подкаталог **Home** и в появившемся списке выберем пункт **Add - > View...** (Представление). В окне добавления нового представления предлагается настроить целый ряд опций:



Разберем все эти настройки:

- **View Name**: имя нового представления. После создания ему автоматически будет присваиваться расширение cshtml.
- **Template**: шаблон нового представления. Мы можем выбрать из следующего списка шаблонов:

Типы шаблонов представления

  - **Empty (without model)**: создается пустое представление с начальной разметкой

  - **Empty**: также создается пустое представление, но теперь ниже мы можем выбрать модель, которая будет подключена в представлении с помощью директивы @model
  - **Create**: генерируется представление с формой для создания новых объектов модели. В этой форме для каждого свойства модели создается отдельное поле
  - **Delete**: генерируется представление с формой для удаления объектов модели. В этой форме для каждого свойства модели создается отдельное поле
  - **Details**: генерируется представление, которое отображает значения всех свойств модели
  - **Edit**: генерируется представление с формой для редактирования имеющихся объектов модели. В этой форме для каждого свойства модели создается отдельное поле
  - **List**: создается представление, которое в таблице отображает все объекта из списка моделей. Для генерации списка объектов в данное представление необходимо передавать из метода контроллера значение типа IEnumerable<Тип_модели>. Представление также содержит ссылки на методы для выполнения операций создания/правки/удаления.
- **Model class**: при выборе одной из предыдущих опций, кроме опции Empty (without model), нам становится доступно это поле, в котором мы можем указать модель для типизации представления. Такое представление будет считаться строго типизированным, то есть привязанным к одному классу модели
- **Data context class**: также при выборе одной из предыдущих опций, кроме опции Empty (without model), нам становится доступно это поле, в котором мы мжем выбрать класс контекста данных
- **Create as a partial view**: позволяет создать частичное представление
- **Reference Script Libraries**: эта опция показывает, будет ли представление автоматически подключать стандартный набор библиотек jQuery и прочих файлов JavaScript.
- **Use a layout page**: эта опция указывает, будет ли использоваться мастер-страница или представление будет самодостаточным. После установки этой опции нам станет доступным нижнее поле, в котором можно выбрать мастер-страницу. Для движка Razor указание мастер-страницы не является обязательным, если вы собираетесь использовать мастер-страницу, определенную по умолчанию в файле _ViewStart.cshtml. Однако, если вы хотите переопределить мастер-страницу, то можете воспользоваться этой опцией.

Пути к файлам представлений

Все добавляемые представления, как правило, группируются по контроллерам в соответствующие папки в каталоге **Views**. Представления, которые относятся к методам контроллера **Home**, будут находиться в проекте в папке **Views/Home**. Однако при необходимости мы сами можем создать в каталоге **Views** папку с произвольным именем, где будем хранить дополнительные представления, необязательно связанные с определенными методами контроллера.

Чтобы произвести **рендеринг** представления в выходной поток, используется метод **View()**. Если в этот метод не передается имени представления, то по умолчанию приложение будет работать с тем представлением, имя которого совпадает с именем метода действия. Например, следующий метод действия будет обращаться к представлению **Index.cshtml**:

```
public ActionResult Index()
        {
            IEnumerable<Book> books = db.Books;
            ViewBag.Books = books;
            return View();
        }
```

Указав путь к представлению явным образом, мы можем переопределить настройки по умолчанию:

```
public ActionResult Index()
        {
            IEnumerable<Book> books = db.Books;
            ViewBag.Books = books;
            return View("~/Views/Some/SomeView.cshtml");
        }
```

Синтаксис Razor

Стандартное представление очень похоже на обычную веб-страницу с кучей кода **html**. Однако оно также имеет вставки кода на **C#**, которые предваряются знаком **@**. Этот знак используется движком представлений **Razor** для перехода к коду на языке **C#**. Чтобы понять суть работы движка **Razor** и его синтаксиса, вначале посмотрим, что представляют из себя движки представлений.

Движок представлений

При вызове метода **View** контроллер не производит рендеринг представления и не генерирует разметку **html**. Контроллер только готовит данные и выбирает, какое представление надо возвратить в качестве объекта **ViewResult**. Затем уже объект **ViewResult** обращается к движку представления для рендеринга представления в выходной результат.

Если ранее предыдущие версии **ASP.NET MVC** и **Visual Studio** по умолчанию поддерживали два движка представлений - движок **Web Forms** и движок **Razor**, то сейчас **Razor** в силу своей простоты и легкости стал единственным движком по умолчанию. Использование **Razor** позволило уменьшить синтаксис при вызове кода **C#,** сделать сам код более "чистым".

Здесь важно понимать, что **Razor** - это не какой-то новый язык, это лишь способ рендеринга представлений, который имеет определенный синтаксис для перехода от разметки **html** к коду **C#**.

Основы синтаксиса Razor

Использование синтаксиса Razor характеризуется тем, что перед выражением кода стоит знак **@**, после которого осуществляется переход к коду **C#**. Существуют два типа переходов: к выражениям кода и к блоку кода.
Например, переход к выражению кода:

```
<p>@b.Name</p>
```

Razor автоматически распознает, что **Name** - это свойство объекта **b**. Также можно использовать стандартные классы и методы, например, выведем текущее время:

```
<h3>@DateTime.Now.ToShortTimeString()</h3>
```

Применение блоков кода аналогично, только знак **@** ставится перед всем блоком кода, а движок автоматически определяет, где этот блок кода заканчивается:

```
@foreach (var b in ViewBag.Books)
            {
                <tr>
                    <td><p>@b.Name</p></td>
                    <td><p>@b.Author</p></td>
                    <td><p>@b.Price</p></td>
                </tr>
            }
```

Более того мы можем создавать блоки кода в представлении, создавать там переменные так же, как и в файле кода **C#**:

```
@{
    string head = "Привет мир!!!";
    head = head + " Добро пожаловать на сайт!";
}
<h3>@head</h3>
```

Строго типизированные представления

В предыдущих примерах для передачи информации из контроллера в представление использовался объект **ViewBag**:

```
@foreach (var b in ViewBag.Books)
            {
                <tr>
                    <td><p>@b.Name</p></td>
                    <td><p>@b.Author</p></td>
                    <td><p>@b.Price</p></td>
                    <td><p><a href="/Home/Buy/@b.Id">Харид
килиш</a></p></td>
                </tr>
            }
```

Здесь мы получаем доступ к элементам коллекции, заключенной в **ViewBag.Books**, с помощью переменной с ключевым словом **var** - то есть тип переменной у нас не задан явно и выводится компилятором. То же самое мы могли бы указать тип модели явно, применив полное имя типа модели:

```
@foreach (BookStore.Models.Book b in ViewBag.Books)
            {
                <tr>
                    <td><p>@b.Name</p></td>
                    <td><p>@b.Author</p></td>
                    <td><p>@b.Price</p></td>
                    <td><p><a href="/Home/Buy/@b.Id">Купить</a></p></td>
                </tr>
            }
```

Хотя примеры с объектом **ViewBag** работают как надо, но есть и другой способ, иногда более предпочтительный, который заключается в использовании **строго типизированных представлений**. Подобные представления позволяют передавать данные не через объект **ViewBag**, а напрямую в представление через параметр метода **View**. Код метода контроллера мог бы выглядеть так:

```
// маълумотлар контекстини ҳосил қиламиз
BookContext db = new BookContext();

public ActionResult Index()
        {
            return View(db.Books);
```

```
        }
```

Теперь, чтобы связать представление с передаваемым параметром, надо добавить в представление директиву **@model** с указанием типа передаваемых данных. Поскольку **books** представляет тип **IEnumerable<Book>**, то представление будет выглядеть так:

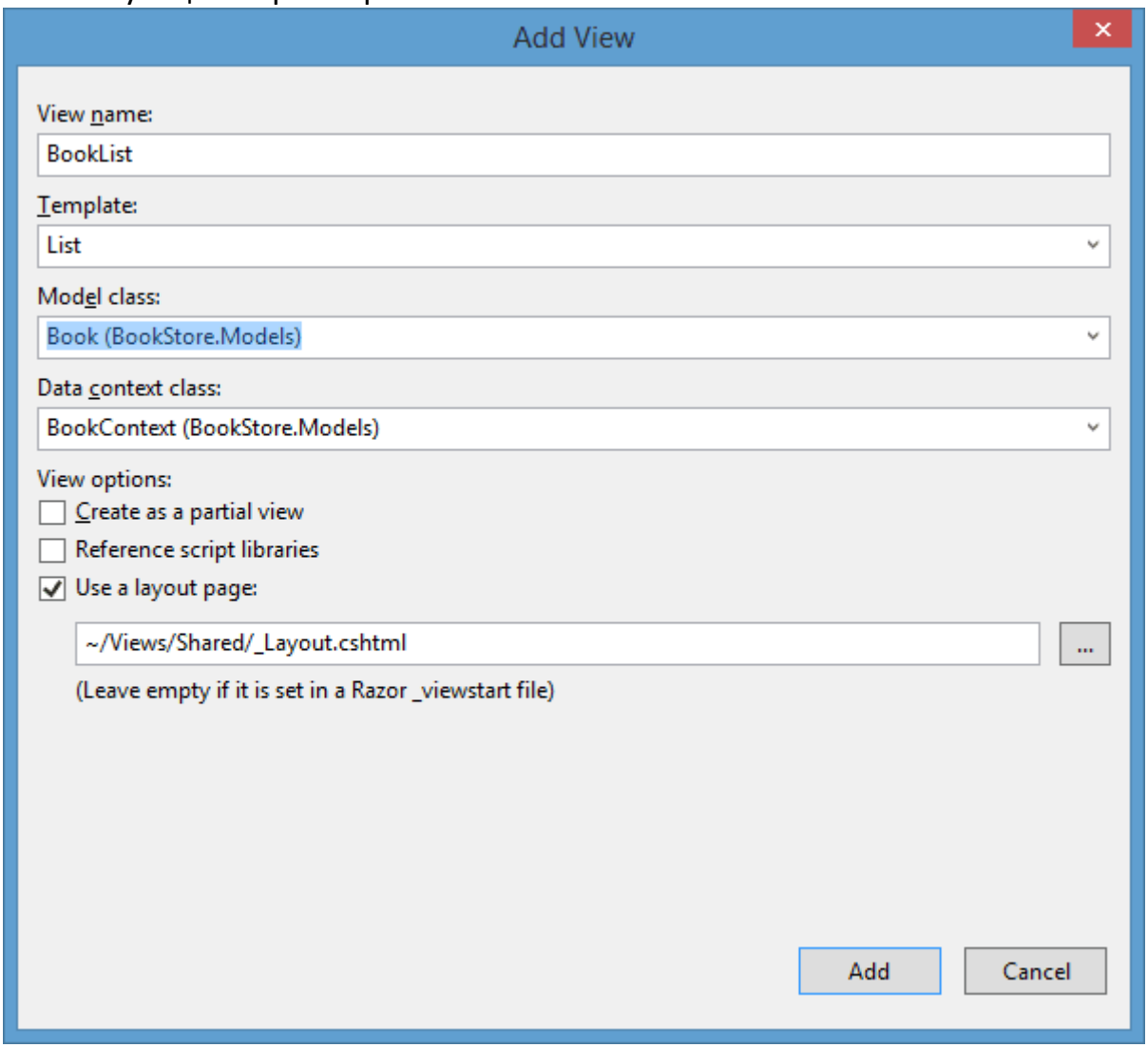
```
@model IEnumerable<BookStore.Models.Book>
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}
<div>
    <h3>Распродажа книг</h3>
    <table>
        <tr class="header">
            <td><p>Название книги</p></td>
            <td><p>Автор</p></td>
            <td><p>Цена</p></td>
            <td></td>
        </tr>
        @foreach (BookStore.Models.Book b in Model)
        {
            <tr>
                <td><p>@b.Name</p></td>
                <td><p>@b.Author</p></td>
                <td><p>@b.Price</p></td>
                <td><p><a href="/Home/Buy/@b.Id">Купить</a></p></td>
            </tr>
        }
    </table>
</div>


</body>
</html>
```

Объект **Model** представляет тип модели, указанной в директиве **@model**, и будет хранить переданные из контроллера данные.

Чтобы не писать полностью и содержит передаваемые в параметре данные. Но чтобы не писать полностью имя типа модели, мы можем импортировать пространство имен в представлении:

```
@using BookStore.Models
@model IEnumerable<Book>
```

Кроме того, мы можем автоматически создать строго типизированное представление, указав в диалоговом окне при создании представления соответствующие параметры:



Для этого в поле **Template** надо выбрать любой другой шаблон, кроме **Empty (without model)**, и после этого указать нужный класс модели и контекста данных. И если мы выберем шаблон **List**, то автоматически сгенерированное представление будет по своему функционалу идентично ранее рассмотренному представлению с выводом книг.

Мастер-страницы

Мастер-страницы используются для создания единообразного, унифицированного вида сайта. По сути мастер-страницы - это те же самые представления, но позволяющие включать в себя другие представления. Например, можно определить на мастер-странице общие для всех остальных представлений элементы, а также подключить общие стили и скрипты.

В итог нам не придется на каждом отдельном представлении прописывать путь к файлам стилей, а потом при необходимости его изменять. А используемые на мастер-страницах заполнители или плейсхолдеры позволят вставить на их место другие представления.

По умолчанию при создании нового проекта **ASP.NET MVC 5** в проект уже добавляется мастер-страница под названием **_Layout.chtml**, которую можно найти в каталоге **Views/Shared**. В приложении из второй главы мы ее изменили следующим образом:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>@ViewBag.Title</title>
    <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet"
type="text/css" />
</head>

<body>
    <nav>
        <ul class="menu">
            <li>@Html.ActionLink("Бош сахифа", "Index", "Home")</li>
        </ul>
    </nav>
    @RenderBody()
</body>
</html>
```

На первый взгляд это обычное представление за одним исключением: здесь используется метод **@RenderBody()**, который является заместителем и на место которого потом будут подставляться другие представления, использующие данную мастер-страницу.

В итоге мы сможем легко установить для представлений веб-приложения единообразный стиль.

Чтобы в представлении использовать мастер-страницу, нам надо в секции **Layout** указать путь к нужной мастер-странице. Например, в представлении Index.cshtml можно определить мастер-страницу так:

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

Если мы не используем мастер-страницу, то указываем **Layout = null;**. Также необязательно для всех представлений использовать одну и ту же мастер-страницу. Можно определить несколько мастер-страниц, например, одну для форума, одну для блога и т.д., и в зависимости от раздела сайта подключать нужную. Добавляются в проект они также как и обычные представления.

Мастер-страница может иметь несколько секций, куда представления могут поместить свое содержимое. Например, добавим к мастер-странице секцию **footer**:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>@ViewBag.Title</title>
    <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet"
type="text/css" />
</head>

<body>
    <nav>
        <ul class="menu">
            <li>@Html.ActionLink("Бош сахифа", "Index", "Home")</li>
        </ul>
    </nav>
    @RenderBody()
    <footer>@RenderSection("Footer")</footer>
</body>
</html>
```

Теперь при запуске предыдущего представления **Index** мы получим ошибку, так как секция **Footer** не определена. По умолчанию представление должно передавать содержание для каждой секции мастер-страницы. Поэтому добавим вниз представления **Index** секцию **footer**. Это мы можем сделать с помощью выражения **@section**:

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

...

@section Footer {
    Все права защищены. Syte Corp. 2015.
}
```

Но при таком подходе, если у нас есть куча представлений, и мы вдруг захотели определить новую секцию на мастер-странице, нам придется изменить все имеющиеся представления, что не очень удобно. В этом случае мы можем воспользоваться одним из вариантов гибкой настройки секций.

Первый вариант заключается в использовании перегруженной версии метода **RenderSection**, которая позволяет указать, что данную секцию не обязательно определять в представлении. Чтобы отметить секцию **Footer** в качестве необязательной, надо передать в метод в качестве второго параметра значение false:

```
<footer>@RenderSection("Footer", false)</footer>
```

Второй вариант позволяет задать содержание секции по умолчанию, если данная секция не определена в представлении:

```
<footer>
    @if (IsSectionDefined("Footer"))
    {
        @RenderSection("Footer")
    }
    else
    {
        <span>Содержание элемента footer по умолчанию.</span>
    }
</footer>
```

ViewStart

Если у нас в проекте пара-тройка представлений, мы легко можем изменить вручную для каждого описание мастер-страницы в секции **Layout**, если, например, мы решим использовать другую мастер-страницу. Но если у нас много представлений, то это делать будет не очень удобно.

Для более гибкой настройки представлений предназначена страница **_ViewStart.cshtml**. Код этой страницы выполняется до кода любого из представлений, расположенных в том же каталоге. Данный файл последовательно применяется к каждому представлению, находящемуся в одном каталоге.

При создании проекта **ASP.NET MVC 5** в каталог **Views** уже по умолчанию добавляется файл **_ViewStart.cshtml**. Этот файл определяет мастер-страницу, используемую по умолчанию:

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

Данный код выполняется до любого другого кода, определенного в представлении, поэтому из других представлений мы можем удалить секцию **Layout**. Если же представление должно использовать другую мастер-страницу, то мы просто переопределяем свойство **Layout**, дописывая его определение в начало представления.

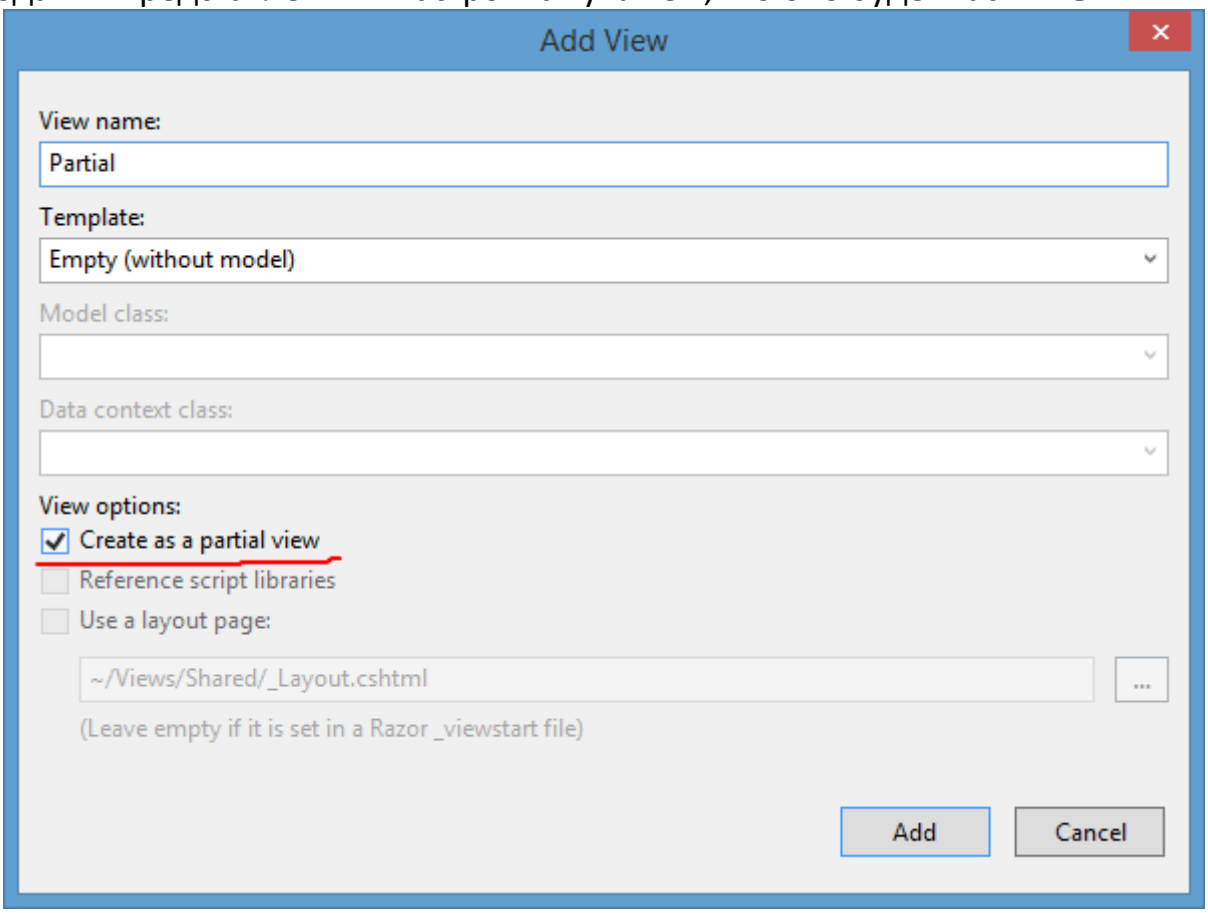
Частичные представления

Кроме обычных представлений метод действия может также возвращать частичные представления. Их отличительной особенностью является то, что их можно встраивать в другие обычные представления. Частичные представления могут использоваться также как и обычные, однако наиболее удобной областью их использования является рендеринг результатов **AJAX**-запроса.

За рендеринг частичных представлений отвечает объект **PartialViewResult**, который возвращается методом **PartialView**. Итак, определим в контроллере новое действие **Partial**:

```
public ActionResult Partial()
        {
            ViewBag.Message = "Это частичное представление.";
            return PartialView();
        }
```

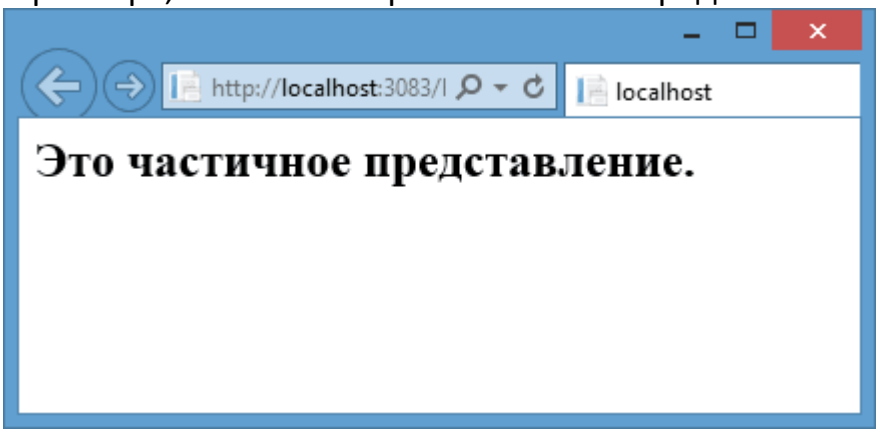
Теперь добавим новое представление **Partial.cshtml**. Для этого при создании представления в настройках укажем, что оно будет частичным:



После этого в проект будет добавлен пустой файл частичного представления. По своему содержанию оно похоже на обычное представление, только для него нельзя определить мастер-страницу. Итак, добавим в частичное представление следующую строку:

```
<h2>@ViewBag.Message</h2>
```

После этого мы можем к обратиться к действию **Partial**, как к обычному действию контроллера, и оно нам вернет частичное представление:



Но смысл применения частичных представлений состоит не в этом, иначе они бы никак не отличались от обычных. Поэтому теперь встроим его в какое-нибудь другое представление. Для этого нам надо применить в любом месте обычного представления хелпер **Html.Partial**:

@Html.Partial("Partial")

В данном случае в качестве параметра мы указываем имя частичного представления без расширения файла. Но в этом случае надо учитывать, что передать **ViewBag.Message** из метода **Partial**, как в предыдущем примере, мы уже не сможем. И если мы все таки хотим передать его в частичное представление, то нам надо будет передать его из метода контроллера, связанного с главным представлением. То есть если мы используем частичное представление в представлении **Index.cshtml**, то в методе **Index** мы можем написать:

```
public ActionResult Index()
        {
            ViewBag.Message = "Это вызов частичного представления из
обычного";
            return View();
        }
```

И, таким образом, сообщение во **ViewBag.Message** будет передано как главному, так и частичному представлению.

Кроме хелпера **Html.Partial** частичное представление можно встроить с помощью другого хелпера - **Html.RenderPartial**. Этот хелпер также принимает имя представления, только он используется не в строчных выражениях кода **Razor**, а в блоке кода - то есть обрамляется фигурными скобками:
@{Html.RenderPartial("Partial");}

Еще одно отличие между двумя способами заключается в том, что **Html.RenderPartial** напрямую пишет вывод в выходной поток, поэтому может работать чуть быстрее, чем **Html.Partial** .

Также как и в случае с обычными представлениями, мы можем создавать строго типизированные частичные представления, указав в шапке файла директиву **@model**:

```
@model IEnumerable<string>
<h2>Список стран</h2>
<ul>
    @foreach (string t in Model)
    {
        <li>@t</li>
    }
</ul>
```

Тогда мы можем вызвать это представление так:

```
@Html.Partial("Partial", new string[] { "Russia", "USA", "Canada",
"France" })
```

HTML-хелперы

Как мы увидели из прошлых примеров, представления используют разметку **html** для визуализации содержимого. Однако фреймворк **ASP.NET MVC** обладает также таким мощным инструментом как HTML-хелперы, позволяющие генерировать **html-код**.

Строчные хелперы

Строчные хелперы похожи на обычные определения методов на языке **C#**, только начинаются с тега **@helper**. Например, создадим в представлении хелпер для вывода названий книг в виде списка:

```
@helper BookList(IEnumerable<BookStore.Models.Book> books)
{
    <ul>
        @foreach (BookStore.Models.Book b in books)
        {
            <li>@b.Name</li>
        }
    </ul>
}
```

Данный хелпер мы можем определить в любом месте представления. И также в любом месте представления мы можем его использовать, передавая в него объект **IEnumerable<BookStore.Models.Book>**:

```
<h3>Список книг</h3>
@BookList(ViewBag.Books)
<!-- или если используется строго типизированное представление -->
@BookList(Model)
```

Строчные **html**-хелперы удобно использовать, если необходимо создать один метод, который предполагается использовать в представлении многократно. Например:

```
@helper CreateList(string[] all)
{
    <ul>
        @foreach (string s in all)
        {
            <li>@s</li>
        }
    </ul>
}
```

```
@{
    string[] cities = new string[] { "Лондон", "Париж", "Москва" };
}
@{
    string[] countries = new string[] { "Великобритания", "Франция",
"Россия" };
}
<h3>Города</h3>
@CreateList(cities)
<br />
<h3>Страны</h3>
@CreateList(countries)
```

При отсутсвии подобного хелпера, то нам бы пришлось по сути дублировать один и тот же **html-код** для создания списка. Однако этот хелпер еще довольно простой, а если нам приходится создавать по сто раз более сложную, но однотипную разметку **html**, тогда хелперы окажутся еще более полезными.

Но данный подход имеет один недостаток - если хелпер очень объемный, то он может очень сильно захламлять разметку представления. И в этом случае его лучше вынести в отдельный файл кода. Так, перепишем предыдущий пример. Для этого нам надо создать новый класс с методом расширения - то есть таким методом, который расширяет функциональность уже существующих классов. А эти классы указываются в качестве первого параметра метода. Итак, создадим в проекте новую папку **Helpers** и добавим в нее новый класс **ListHelper**:

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

namespace BookStore.Helpers
{
    public static class ListHelper
    {
        public static MvcHtmlString CreateList(this HtmlHelper html,
string[] items)
        {
            TagBuilder ul = new TagBuilder("ul");
            foreach (string item in items)
            {
                TagBuilder li = new TagBuilder("li");
                li.SetInnerText(item);
                ul.InnerHtml += li.ToString();
            }
            return new MvcHtmlString(ul.ToString());
        }
```

```
    }
}
```

В новом классе хелпера определен один статический метод **CreateList**, принимающий в качестве первого параметра объект, для которого создается метод. Так как данный метод расширяет функциональность **html**-хелперов, которые представляет класс **HtmlHelper**, то именно объект этого типа и передается в данном случае в качестве первого параметра. Второй параметр метода **CreateList** - массив строк-значений, которые потом будут выводиться в списке.

В самом методе с помощью объекта **TagBuilder** конструируется стандартный элемент **html** - элемент **ul**. При обходе массива все строковые значения обертываются в тег **li** и добавляются в список. И на выходе возвращается полноценные элемент **ul**.

Класс **TagBuilder** имеет ряд членов, которые можно использовать при таком подходе:

- Свойство **InnerHtml** позволяет установить или получить содержимое тега в виде строки;
- Метод **MergeAttribute (string, string, bool)** позволяет добавить к элементу один атрибут. Для получения всех атрибутов можно использовать коллекцию **Attributes**;
- Метод **SetInnerText(string)** устанавливает текстовое содержимое внутри элемента;
- Метод **AddCssClass(sting)** добавляет класс **css** к элементу

После создания нового хелпера мы его можем использовать в представлении. Перепишем предыдущий пример следующим образом:

```
@{
    string[] cities = new string[] { "Лондон", "Париж", "Москва" };
}
@{
    string[] countries = new string[] { "Великобритания", "Франция",
"Россия" };
}

@using BookStore.Helpers

<h3>Города</h3>
@Html.CreateList(cities)
<br />
<h3>Страны</h3>
<!-- или можно вызвать так -->
@ListHelper.CreateList(Html, countries)
```

Работа с формами

Хотя мы можем сами написать любой требуемый хелпер, но фреймворк **MVC** уже предоставляет большой набор встроенных **html-хелпер**ов, которые позволяют генерировать ту или иную разметку, главным образом, для работы с формами. Поэтому в большинстве случаев не придется создавать свои хелперы, и можно будет воспользоваться встроенными.

Хелпер Html.BeginForm

Для создания форм мы вполне можем использовать стандартные элементы **html**, например:

```
<form method="post" action="/Home/Buy">
    <input type="hidden" value="@ViewBag.BookId" name="BookId" />
    <table>
        <tr>
            <td><p>Введите свое имя </p></td>
            <td><input type="text" name="Person" /> </td>
        </tr>
        <tr>
            <td><p>Введите адрес :</p></td>
            <td><input type="text" name="Address" /> </td>
        </tr>
        <tr>
            <td><input type="submit" value="Отправить" /> </td>
            <td></td>
        </tr>
    </table>
</form>
```

Это обычная **html-форма**, которая по нажатию на кнопку отправляет все введенные данные запросом **POST** на адрес **/Home/Buy**. Встроенный хелпер **BeginForm/EndForm** позволяет создать ту же самую форму:

```
@using (Html.BeginForm("Buy", "Home", FormMethod.Post))
{
    <input type="hidden" value="@ViewBag.BookId" name="BookId" />
    <table>
        <tr>
            <td><p>Введите свое имя </p></td>
            <td><input type="text" name="Person" /> </td>
        </tr>
        <tr>
            <td><p>Введите адрес :</p></td>
            <td><input type="text" name="Address" /> </td>
```

```
        </tr>
        <tr>
            <td><input type="submit" value="Отправить" /> </td>
            <td></td>
        </tr>
    </table>
}
```

Метод **BeginForm** принимает в качестве параметров имя метода действия и имя контроллера, а также тип запроса. Данный хелпер создает как открывающий тег **<form>**, так и закрывающий тег **</form>**. Поэтому при рендеринге представления в выходной поток у нас получится тот же самый **html-код**, что и с применением тега **form**. Поэтому оба способа идентичны.

Здесь есть один момент. Если у нас в контроллере определены две версии одного метода - для методов **POST** и **GET**, например:

```
[HttpGet]
public ActionResult Buy(int id)
        {
            if (id > 2)
            {
                return Redirect("/Home/Index");
            }
            ViewBag.BookId = id;
            return View();

        }

[HttpPost]
public string Buy(Purchase purchase)
        {
            db.Purchases.Add(purchase);
            db.SaveChanges();
            return "Сегодня: " + purchase.Date + "Спасибо," +
purchase.Person + ", за покупку!";
        }
```

То есть фактически вызов страницы с формой и отправка формы осуществляется одним и тем же действием **Buy**. В этом случае можно не указывать в хелпере **Html.BeginForm** параметры:

```
@using(Html.BeginForm())
{
    ..............
}
```

Ввод информации

В предыдущем примере вместе с хелпером **Html.BeginForm** использовались стандартные элементы **html**. Однако набор **html-хелперов** содержит также хелперы для ввода информации пользователем. В **MVC** определен широкий набор хелперов ввода практически для каждого html-элемента. Что выбрать - хелпер или стандартный элементы ввода **html**, уже решает сам разработчик.

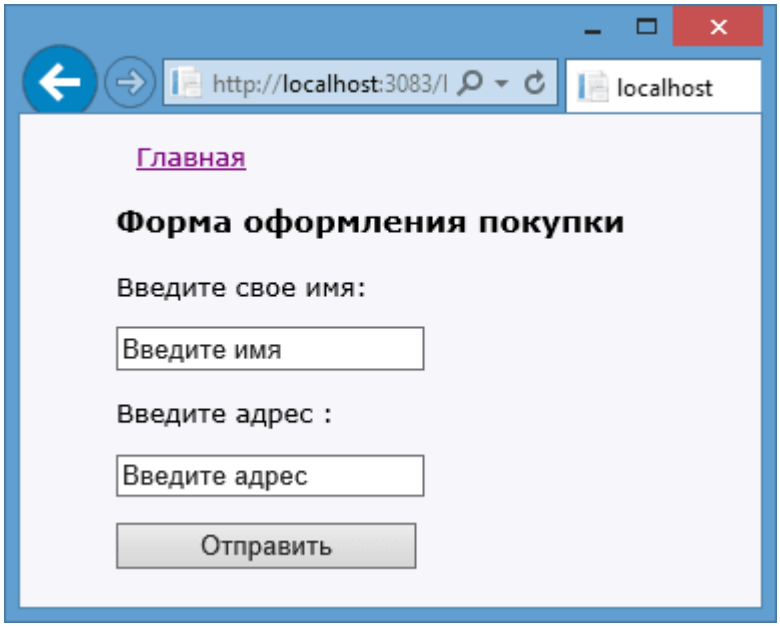
Вне зависимости от типа все базовые **html-хелпер**ы используют как минимум два параметра: первый параметр применяется для установки значений для атрибутов **id** и **name**, а второй параметр - для установки значения атрибута **value**

Html.TextBox

Хелпер **Html.TextBox** генерирует тег **input** со значением атрибута **type** равным **text**. Хелпер **TextBox** используют для получения ввода пользователем информации. Так, перепишем предыдущую форму с заменой полей ввода на хелпер **Html.TextBox**:

```
@using (Html.BeginForm("Buy", "Home", FormMethod.Post))
{
    <input type="hidden" value="@ViewBag.BookId" name="BookId" />
    <p>Введите свое имя: </p>
    @Html.TextBox("Person", "Введите имя")
    <p>Введите адрес :</p>
    @Html.TextBox("Address", "Введите адрес")
    <p><input type="submit" value="Отправить" /></p>
}
```

Мы получим тот же результат:



Html.TextArea

Хелпер **TextArea** используется для создания элемента **<textarea>**, который представляет многострочное текстовое поле. Результатом выражения **@Html.TextArea("text", "привет
 мир")** будет следующая **html-разметка**:

```
<textarea cols="20" id="text" name="text" rows="2">привет <br/> мир
</textarea>
```

Обратите внимание, что хелпер декодирует помещаемое в него значение, в том числе и **html-теги**, (все хелперы декодируют значения моделей и значения атрибутов). Другие версии хелпера **TextArea** позволяют указать число строк и столбцов, определяющих размер текстового поля.

```
@Html.TextArea("text", "привет <br /> мир", 5, 50, null)
```

Этот хелпер сгенерирует следующую разметку:

```
<textarea cols="50" id="text" name="text" rows="5">привет <br /> мир
</textarea>
```

Html.Hidden

В примере с формой мы использовали скрытое поле **input type="hidden"**, вместо которого могли бы вполне использовать хелпер **Html.Hidden**. Так, следующий вызов хелпера:

```
@Html.Hidden("BookId", "2")
```

сгенерирует разметку:

```
<input id="BookId" name="BookId" type="hidden" value="2" />
```

А при передаче переменной из **ViewBag** нам надо привести ее к типу **string**:

```
@Html.Hidden("BookId", @ViewBag.BookId as string)
```

Html.Password

Html.Password создает поле для ввода пароля. Он похож на хелпер **TextBox**, но вместо введенных символов отображает маску пароля. Следующий код:

```
@Html.Password("UserPassword", "val")
```

генерирует разметку:

```
<input id="UserPassword" name="UserPassword" type="password" value="val"
/>
```

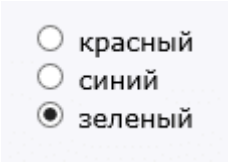
Html.RadioButton

Для создания переключателей применяется хелпер **Html.RadioButton**. Он генерирует элемент **input** со значением **type="radio"**. Для создания группы переключателей, надо присвоить всем им одно и то же имя (свойство name):

```
@Html.RadioButton("color", "red")
<span>красный</span> <br />
@Html.RadioButton("color", "blue")
<span>синий</span> <br />
@Html.RadioButton("color", "green", true)
<span>зеленый</span>
```

Этот код создает следующую разметку:

```
<input id="color" name="color" type="radio" value="red" />
<span>красный</span> <br />
<input id="color" name="color" type="radio" value="blue" />
<span>синий</span> <br />
<input checked="checked" id="color" name="color" type="radio"
value="green" />
<span>зеленый</span>
```



Html.CheckBox

Html.CheckBox может применяться для создания сразу двух элементов. Возьмем, к примеру, следующий код:

```
@Html.CheckBox("Enable", false)
```

Это выражение будет генерировать следующий **HTML**:

```
<input id="Enable" name="Enable" type="checkbox" value="true" />
<input name="Enable" type="hidden" value="false" />
```

То есть кроме собственно поля флажка, еще и генерируется скрытое поле. Зачем оно нужно? Дело в том, что браузер посылает значение флажка только тогда, когда флажок выбран или отмечен. А скрытое поле гарантирует, что для элемента **Enable** будет установлено значение даже, если пользователь не отметил флажок.

Html.Label

Хелпер **Html.Label** создает элемент **<label/>**, а передаваемый в хелпер параметр определяет значение атрибута **for** и одновременно текст на элементе. Перегруженная версия хелпера позволяет определить значение атрибута **for** и текст на метке независимо друг от друга. Например, объявление хелпера **Html.Label("Name")** создает следующую разметку:

```
<label for="Name">Name</label>
```

Элемент **label** представляет простую метку, предназначенную для прикрепления информации к элементам ввода, например, к текстовым полям. Атрибут **for** элемента **label** должен содержать **ID** ассоциированного элемента ввода. Если пользователь нажимает на метку, то браузер автоматически передает фокус связанному с этой меткой элементу ввода.

Html.DropDownList

Хелпер **Html.DropDownList** создает выпадающий список, то есть элемент **<select/>**. Для генерации такого списка нужна коллекция объектов **SelectListItem**, которые представляют элементы списка. Объект **SelectListItem** имеет свойства **Text** (отображаемый текст), **Value**(само значение, которое может не совпадать с текстом) и **Selected**. Можно создать коллекцию объектов **SelectListItem** или использовать хелпер **SelectList**. Этот

хелпер просматривает объекты **IEnumerable** и преобразуют их в последовательность объектов **SelectListItem**.

Так, код `@Html.DropDownList("countires", new SelectList(new string[] {"Russia","USA", "Canada","France"}),"Countries")`
генерирует следующую разметку:

```
<select id="countires" name="countires"><option
value="">Countries</option>
<option>Russia</option>
<option>USA</option>
<option>Canada</option>
<option>France</option>
</select>
```

Теперь более сложный пример. Выведем в список коллекцию элементов **Book**. В контроллере передадим этот список через ViewBag:

```
BookContext db = new BookContext();
public ActionResult Index()
        {
            SelectList books = new SelectList(db.Books, "Author",
"Name");
            ViewBag.Books = books;
            return View();
        }
```

Здесь мы создаем объект **SelectList**, передавая в его конструктор набор значений для списка (**db.Books**), название свойства модели **Book**, которое будет использоваться в качестве значения (**Author**), и название свойства модели **Book**, которое будет использоваться для отображения в списке. В данном случае необязательно устанавливать два разных свойства, можно было и одно установить и для значения и отображения.
Тогда в представлении мы можем так использовать этот **SelectList**:

```
@Html.DropDownList("Author", ViewBag.Books as SelectList)
```

И при рендеринге представления все элементы **SelectList** добавятся в выпадающий список

Html.ListBox

Хелпер **Html.ListBox**, также как и **DropDownList**, создает элемент **<select/>**, но при этом делает возможным множественное выделение элементов (то есть для атрибута **multiple** устанавливается значение **multiple**).

Для создания списка, поддерживающего множественное выделение, вместо **SelectList** можно использовать класс **MultiSelectList**:

```
@Html.ListBox("countires", new MultiSelectList(new string[] { "Россия", "США", "Китай", "Индия" }))
```

Этот код генерирует следующую разметку:

```
<select length="9" id="countries" multiple="multiple" name="countires">
  <option>Россия</option>
  <option>США</option>
  <option>Китай</option>
  <option>Индия</option>
</select>
```

С передачей одиночных значений на сервер все понятно, но как передать множественные значения? Допустим, у нас есть следующая форма:

```
@using (Html.BeginForm())
        {
        @Html.ListBox("countries",
            new MultiSelectList(new string[] { "Россия", "США", "Китай", "Индия" }))
        <p><input type="submit" value="Отправить" /></p>
        }
```

Тогда метод контроллера мог бы получать эти значения следующим образом:

```
[HttpPost]
public string Index(string[] countries)
        {
            string result = "";
            foreach (string c in countries)
            {
                result += c;
                result += ";";
            }
            return "Вы выбрали: " + result;
      }
```

Форма с несколькими кнопками

Как правило, на форме есть только одна кнопка для отправки. Однако в определенных ситуациях может возникнуть потребность, использовать более одной кнопки. Например, есть поле для ввода значения, а две кнопки указывают, надо это значение удалить или, наоборот, добавить:

```
@using (Html.BeginForm("MyAction", "Home", FormMethod.Post))
        {
            <input type="text" name="product" /><br />
            <button name="action" value="add">Добавить</button>
            <button name="action" value="delete">Удалить</button>
        }
```

Самое простое решение состоит в том, что для каждой кнопки устанавливается одниковое значение атрибута **name**, но разное для атрибута **value**. А метод, принимающий форму, может выглядеть следующим образом:

```
[HttpPost]
public string MyAction(string product, string action)
        {
            string res="";
            if (action == "add")
            {
                res="add";
            }
            else if (action == "delete")
            {
                res = "deleted";
            }
            // остальной код метода
            return res;
        }
```

И с помощью условной конструкции в зависимости от значения параметра **action**, который хранит значение атрибута **value** нажатой кнопки, производятся определенные действия.

Строго типизированные хелперы

Кроме базовых хелперов в **ASP.NET MVC** имеются их двойники - строго типизированные хелперы. Этот вид хелперов принимает в качестве параметра лямбда-выражение, в котором указывается то свойство модели, к которому должен быть привязан данный хелпер. Важно учитывать, что строго типизированные хелперы могут использоваться только в строго типизированных представлениях, а тип модели, которая передается в хелпер, должен быть тем же самым, что указан для всего представления с помощью директивы **@model**.

Посмотрим на примере. Во второй главе мы использовали модель **Purchase** для оформления покупки книги:

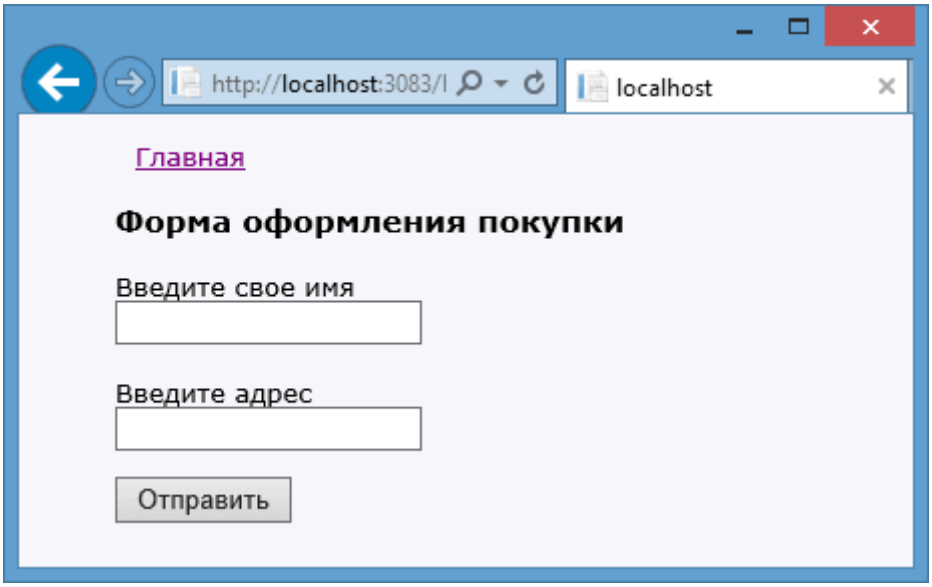
```
public class Purchase
    {
        // харид ID си
        public int PurchaseId { get; set; }
        // харидор исми ва фамилияси
        public string Person { get; set; }
        // харидор манзили
        public string Address { get; set; }
        // китоб ID си
        public int BookId { get; set; }
        // хариф санаси
        public DateTime Date { get; set; }
    }
```

И для ее использования применялась следующая форма:

```
<form method="post" action="">
     <input type="hidden" value="@ViewBag.BookId" name="BookId" />
     <table>
     <tr>
       <td><p>Введите свое имя </p></td>
       <td><input type="text" name="Person" /> </td>
     </tr>
     <tr>
       <td><p>Введите адрес :</p></td>
       <td>
         <input type="text" name="Address" />
       </td>
     </tr>
     <tr>
     <td><input type="submit" value="Отправить" /> </td>
     <td></td>
     </tr>
     </table>
</form>
```

Перепишем этот пример с использованием хелперов:

```
@model BookStore.Models.Purchase
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<div>
    <h3>Форма оформления покупки</h3>
    @using (Html.BeginForm("Buy", "Home", FormMethod.Post))
    {
        @Html.HiddenFor(m => m.BookId)
        @Html.LabelFor(m => m.Person, "Введите свое имя")
        <br />
        @Html.TextBoxFor(m => m.Person)
        <br /><br />
        @Html.LabelFor(m => m.Address, "Введите адрес")
        <br />
        @Html.TextBoxFor(m => m.Address)
        <p><input type="submit" value="Отправить" /></p>
    }
</div>
```



Строго типизированный хелпер похож на обычный, только в конце прибавляется суффикс **For**: **LabelFor**. Так как строго типизированные хелперы могут использоваться только в строго типизированных представлениях, то вначале представления указываем модель, которая будет использоваться:

```
@model BookStore.Models.Purchase
```

То есть, в вызове `@Html.TextBoxFor(m => m.Person)`
параметр **m** представляет переменную модели **Purchase**. А лямбда-выражение **m=>m.Person** указывает, что данный хелпер будет генерировать текстовое поле для свойства **Person**.
Таким образом, хелпер `@Html.TextBoxFor(m => m.Person)` сгенерирует текстовое поле `<input id="Person" name="Person" type="text" value="" />`.

Для каждого базового встроенного хелпера имеется свой строго типизированный хелпер:

- **Html.CheckBoxFor**
  Выражение `@Html.CheckBoxFor(m=>m.Enable, false)` создает разметку:
  ```
  <input id="Enable" name="Enable" type="checkbox" value="true" />
  <input name="Enable" type="hidden" value="false" />
  ```

- **Html.HiddenFor**
  Выражение `@Html.HiddenFor(m=> m.Name)` создает разметку:
  ```
  <input id="Name" name="Name" type="hidden"
  value="[значение_m.Name]" />
  ```

- **Html.LabelFor**
  Хелпер `@Html.LabelFor(m => m.Name,"Имя")` генерирует разметку:
  ```
  <label for="Name">Имя</label>
  ```

- **Html.PasswordFor**
  Хелпер `@Html.PasswordFor(m => m.Password)` оздает разметку:
  ```
  <p><input type="submit" value="Отправить" /></p>
  <p><input id="Password" name="Password" type="password" /></p>
  ```

- **Html.RadioButtonFor**
  `@Html.RadioButtonFor(m => m.Option, "val")` генерирует разметку:
  ```
  <input id="Option" name="Option" type="radio" value="val" />
  ```

- **Html.TextBoxFor**
  Выражение `@Html.TextBoxFor(m => m.Name)` создает разметку:
  ```
  <input id="Name" name="Name" type="text" />
  ```

- **Html.TextAreaFor**
  Хелпер `@Html.TextAreaFor(m => m.Name,10,9,null)` генерирует код:
  ```
  <textarea cols="9" id="Name" name="Name" rows="10" ></textarea>
  ```

Модели

Модели и БД

Все сущности в приложении принято выделять в отдельные модели. В зависимости от поставленной задачи и сложности приложения можно выделить различное количество моделей. Так, в тестовом приложении из второй главы использовались две модели - класс для книги и класс для покупки книги.

Модели представляют собой простые классы и располагаются в проекте в каталоге **Models**. Модели описывают логику данных. Например, модель представляющая книгу и ее покупку:

```
public class Book
    {
        // китоб ID си
        public int Id { get; set; }
        // китоб номи
        public string Name { get; set; }
        // китоб муаллифи
        public string Author { get; set; }
        // нархи
        public int Price { get; set; }
    }
public class Purchase
    {
        // харид ID си
        public int PurchaseId { get; set; }
        // харидор исми ва фамилияси
        public string Person { get; set; }
        // харидор манзили
        public string Address { get; set; }
        // китоб ID си
        public int BookId { get; set; }
        // хариф санаси
        public DateTime Date { get; set; }
    }
```

Модель необязательно состоит только из свойств, кроме того, она может иметь конструктор, какие-нибудь вспомогательные методы. Но главное не перегружать класс модели и помнить, что его предназначение - описывать данные. Манипуляции с данными и бизнес-логика - это больше сфера контроллера.

Данные моделей, как правило, хранятся в базе данных. Для работы с базой данных очень удобно пользоваться фреймворком **Entity Framework**, который позволяет абстрагироваться от написания **sql-запросов**, от строения базы данных и полностью сосредоточиться на логике приложения.

Если при создании проекта **MVC 5** вы выберете в качестве типа аутентификации "**No Authentication**", то после создания проекта в его надо будет подключить **Entity Framework** через пакетный менеджер **NuGet**, как описывалось во второй главе.

В качестве альтернативы **NuGet** можно использовать консоль пакетного менеджера. Для этого в меню **Visual Studio** выберем **View -> Other Windows -> Package Manager Console**. После этого внизу студии откроется консоль пакетного менеджера. В ней введем такую команду:

PM> Install-Package Entity Framework -Version 6.0.2

После ввода команды будет загружен и установлен пакет **Entity Framework**. Иногда этой консолью предпочтительнее пользоваться при установке пакетов, чем менеджером **NuGet**, так как менеджер **NuGet** может немного опаздывать за выпуском последних версий пакетов. Либо наоборот, нам надо установить пакеты более ранней версии, а **NuGet** может предложить только текущую версию.

Entity Framework поддерживает подход "Code first", который предполагает сохранение или извлечение информации из **БД** на **SQL Server** без создания схемы базы данных или использования дизайнера в **Visual Studo**. Наоборот, мы создаем обычные классы, а **Entity Framework** уже сам определяет, как и где сохранять объекты этих классов.

Для подключения к базе данных через **Entity Framework**, нам нужен посредник - **контекст данных**. Контекст данных представляет собой класс, производный от класса **DbContext**. Контекст данных содержит одно или несколько свойств типа **DbSet<T>**, где **T** представляет тип объекта, хранящегося в базе данных. Например, создадим контекст данных для работы с вышеприведенными моделями:

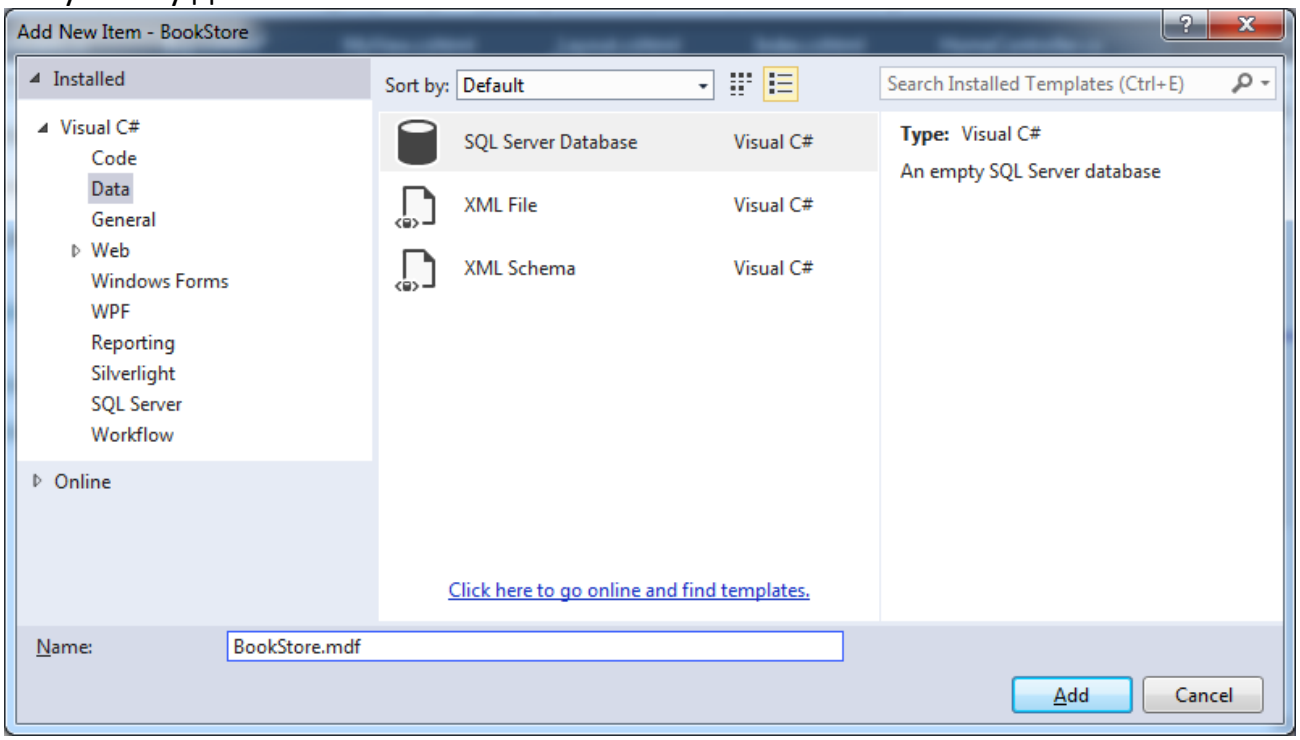
```
using System;
using System.Collections.Generic;
using System.Web;
using System.Data.Entity;
namespace BookStore.Models
{
    public class BookContext : DbContext
    {
        public DbSet<Book> Books { get; set; }
        public DbSet<Purchase> Purchases { get; set; }
    }
}
```

С помощью свойств **Books** и **Purchases** мы получаем доступ к данным соответствующих моделей, которые хранятся в базе данных.

Подключение к базе данных

Для хранения данных приложению нужна база данных. Мы можем использовать различные **СУБД**, но, как правило, в качестве базы данных в связке с **ASP.NET MVC** используется база данных **MS SQL Server**, на примере которого мы и посмотрим весь процесс создания **БД** и подключения к ней.

Мы можем создать базу данных прямо в проекте, либо же создать ее на сервере **MS SQL**. Для хранения баз данных проекте предназначена папка **App_Data**. Итак, нажмем на папку **App_Data** правой кнопкой мыши и в появившемся контекстном меню выберем **Add-> New Item...**. В появившемся окне добавления нового элемента выберем **SQL Server Database** и назовем новую базу данных **Bookstore.mdf**:



После этого в папку папку **App_Data** будет добавлена база данных, и мы можем начинать с ней работать - добавлять таблицы и данные. Но перед этим посмотрим, что у нас будет хранится в **БД**. Для работы с **БД** возьмем модели из предыдущих глав - модель **Book**:

```
public class Book
    {
        // китоб ID си
        public int Id { get; set; }
        // китоб номи
        public string Name { get; set; }
        // китоб муаллифи
        public string Author { get; set; }
        // нархи
```

```
        public int Price { get; set; }
    }
```

И модель **Purchase**:

```
public class Purchase
    {
        // харид ID си
        public int PurchaseId { get; set; }
        // харидор исми ва фамилияси
        public string Person { get; set; }
        // харидор манзили
        public string Address { get; set; }
        // китоб ID си
        public int BookId { get; set; }
        // хариф санаси
        public DateTime Date { get; set; }
    }
```

Пусть для работы с ними определен следующий контекст данных:

```
public class BookContext : DbContext
    {
        public DbSet<Book> Books { get; set; }
        public DbSet<Purchase> Purchases { get; set; }
    }
```

Чтобы связать приложение, контекст данных и БД, добавим в файл **web.config** строку подключения к этой базе данных. Для этого после секции **configSections** вставим следующую секцию:

```
<connectionStrings>
    <add name="BookContext" connectionString="Data
Source=(LocalDB)\v11.0;AttachDbFilename='|DataDirectory|\Bookstore.mdf';I
ntegrated Security=True"
   providerName="System.Data.SqlClient" />
</connectionStrings>
```

Обратите внимание, что значени атрибута **name** (name="BookContext") в качестве значения имеет название контекста данных.

Есть также второй способ. Например, если у нас строка подключения имеет другое название, например **<add name="DefaultConnection"...,** и мы не хотим его менять, то в этом случае для связи с контекстом данных мы можем передать название в конструкор базового класса:

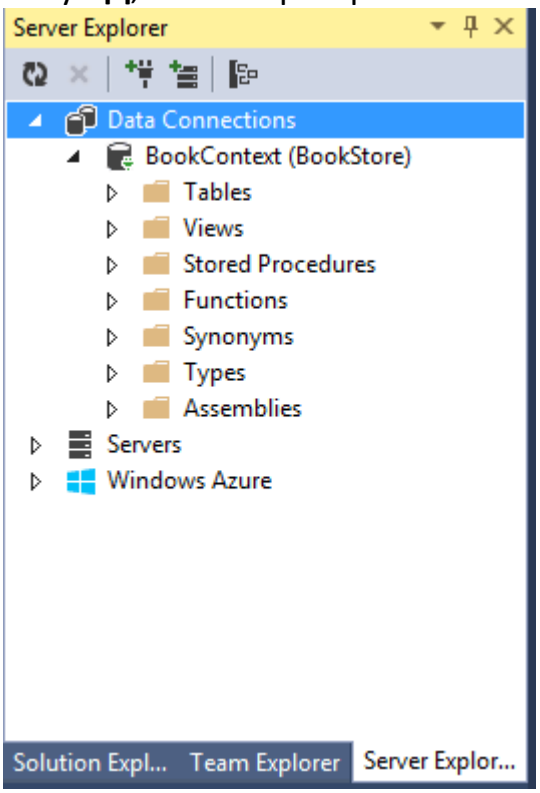
```
public class BookContext : DbContext
    {
```

```
        public BookContext()
            : base("DefaultConnection")
        { }
        public DbSet<Book> Books { get; set; }
        public DbSet<Purchase> Purchases { get; set; }
    }
```

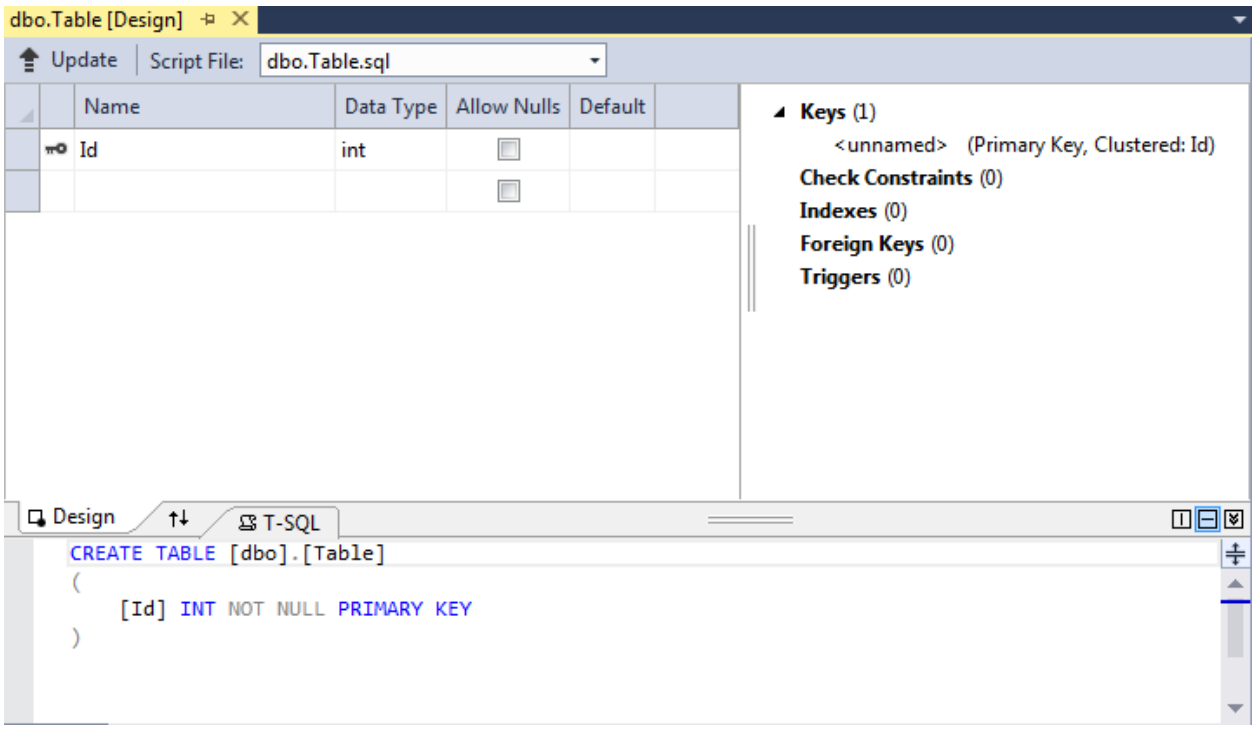
В **Visual Studio 2013** мы можем использовать **LocalDB**. **LocalDB** представляет облегченную версию движка баз данных **SQL Sever Express**, которая специально нацелена на разработчиков. Поэтому в данном случае в качестве источника данных указываем **(LocalDB)\v11.0**.

Использование подстановки **|DataDirectory|** позволяет опустить полный физический путь к базе данных, которая хранится в папке **App_Data**.

Теперь создадим сами таблицы. Для этого перейдем в окно **Server Explorer** и раскроем в нем узел **Data Connections**. Далее мы увидим узел **BookContext**, который представляет нашу **БД**, и также раскроем его:

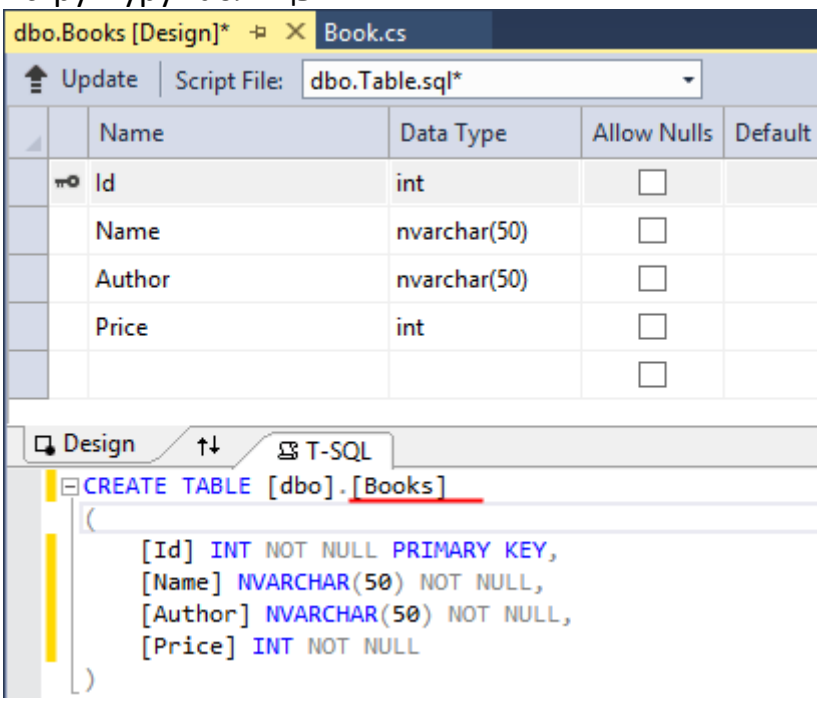


Добавим первую таблицу. Для этого нажмем правой кнопкой мыши на узел **Tables** и в появившемся меню выберем пункт **Add New Table**. После этого отобразится окно дизайнера таблицы, в котором надо определить названия и типы столбцов новой таблицы.

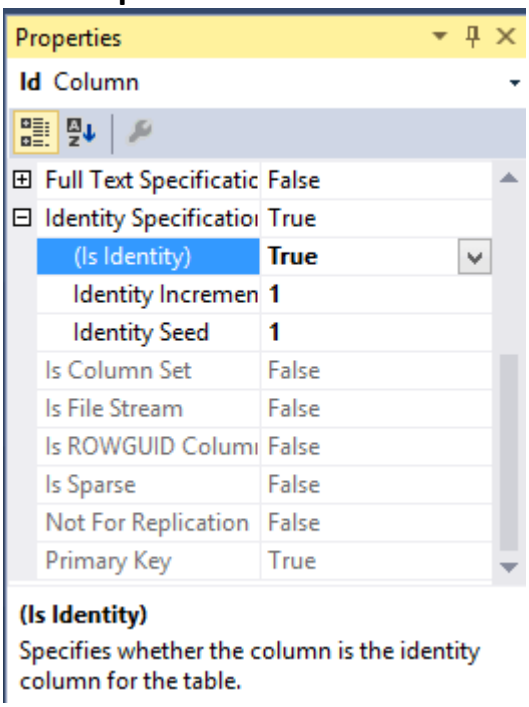


По соглашениям о наименованиях таблицы при работе с **Entity Framework** названия таблиц должны соответствовать имени модели. Например, если модель называется **Book**, то таблица будет называться **Books**. А **Entity Framework** автоматически распознает, что таблица **Books** соответствует классу **Book**.

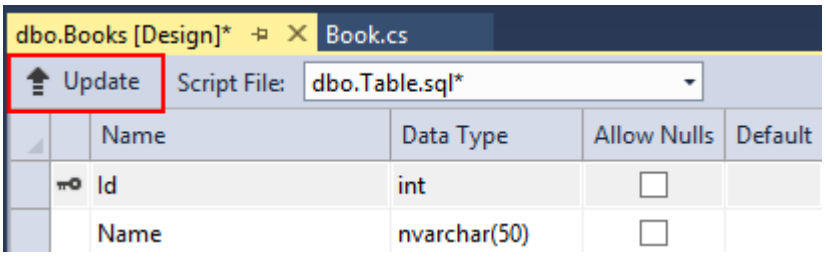
Итак, создадим структуру таблицы:



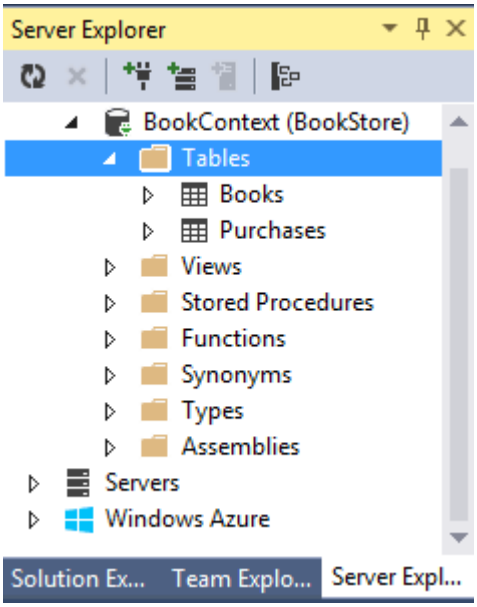
Также поскольку предполагается, что столбец **Id** будет инкрементироваться с добавлением каждого нового объекта, то установим для него автоинкремент в окне **Properties**:



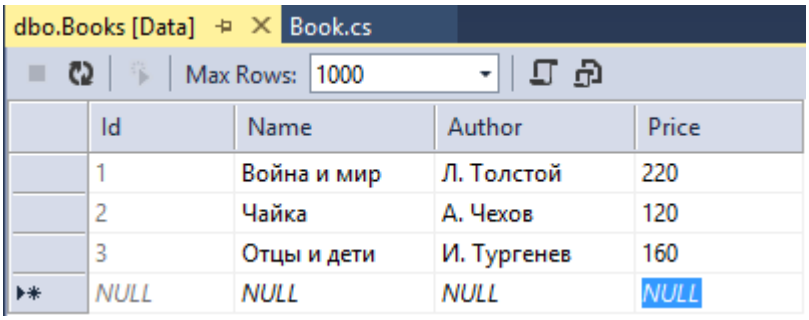
И последний шаг - генерация базы данных. Для этого нажмем на кнопку Update:



В появившемся диалоговом окне нажмем на кнопку **Update Database**. После этого в нашу базу данных добавится новая таблица. Подобным образом определим таблицу **Purchases** для модели **Purchase**:



Добавим в дизайнере баз данных в таблицу **Books** несколько записей. Для этого нажмем в окне **Server Explorer** на узел **Books** и выберем в появившемся списке пункт **Show Table Data**. Добавим, к примеру, следующий набор записей:



	Id	Name	Author	Price
	1	Война и мир	Л. Толстой	220
	2	Чайка	А. Чехов	120
	3	Отцы и дети	И. Тургенев	160
▶*	NULL	NULL	NULL	NULL

Теперь мы можем получить эти данные в контроллере **Home** и передать их в представление:

```
BookContext db = new BookContext();
public ActionResult Index()
        {
            return View(db.Books);
        }
```

Вывод данных в представлении Index.cshtml:

```
@model IEnumerable<BookStore.Models.Book>
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<div>
```

```
    <h3>Распродажа книг</h3>
    <table>
        <tr class="header">
            <td><p>Название книги</p></td>
            <td><p>Автор</p></td>
            <td><p>Цена</p></td>
            <td></td>
        </tr>
        @foreach (BookStore.Models.Book b in Model)
        {
            <tr>
                <td><p>@b.Name</p></td>
                <td><p>@b.Author</p></td>
                <td><p>@b.Price</p></td>
                <td><p><a href="/Home/Buy/@b.Id">Купить</a></p></td>
            </tr>
        }
    </table>
</div>
```

Закрытие подключения

Чтобы наверняка быть уверенным, что подключение к базе данных закрыто, следует вызывать метод **Dispose** у контекста данных:

```
protected override void Dispose(bool disposing)
        {
            db.Dispose();
            base.Dispose(disposing);
        }
```

Это переопределенная версия метода **Dispose** контроллера, которая вызывается при уничтожении объекта контроллера. В нее помещается вызов **db.Dispose()**, который уничтожает все связанные с контекстом данных ресурсы и подключения.

Шаблонные хелперы

Кроме базовых **html-хелперов**, рассмотренных в прошлой главе и генерирующих определенные элементы разметки **html**, фреймворк **ASP.NET MVC** также имеет **шаблонные (или шаблонизированные) хелперы**. В отличие от рассмотренных в прошлой главе **html**-хелперов они не генерируют определенный элемент **html**. Шаблонные хелперы смотрят на свойство модели и генерируют тот элемент **html**, который наиболее подходит данному свойству, исходя из его типа и метаданных.

В **ASP.NET MVC** имеются следующие шаблонные хелперы:

- **Display**
  Создает элемент разметки для отображения значения указанного свойства модели: Html.Display("Name")
- **DisplayFor**
  Строго типизированный аналог хелпера Display: Html.DisplayFor(m => m.Name)
- **Editor**
  Создает элемент разметки для редактирования указанного свойства модели: Html.Editor("Name")
- **EditorFor**
  Строго типизированный аналог хелпера Editor: Html.EditorFor(m => m.Name)
- **DisplayText**
  Создает выражение для указанного свойства модели в виде простой строки: Html.DisplayText("Name")
- **DisplayTextFor**
  Строго типизированный аналог хелпера DisplayText: Html.DisplayTextFor(m => m.Name)

Это были одиночные хелперы, которые генерируют разметку только для одного свойства модели. Но кроме них во фреймворке также есть еще несколько шаблонов, которые позволяют создать разом все поля для всех свойств модели:

- **DisplayForModel**
  Создает поля для чтения для всех свойств модели: Html.DisplayForModel()
- **EditorForModel**
  Создает поля для редактирования для всех свойств модели: Html.EditorForModel()

Например, определим в контроллере некоторое действие **BookView**, которое по **Id** будет выводить информацию об определенной книге:

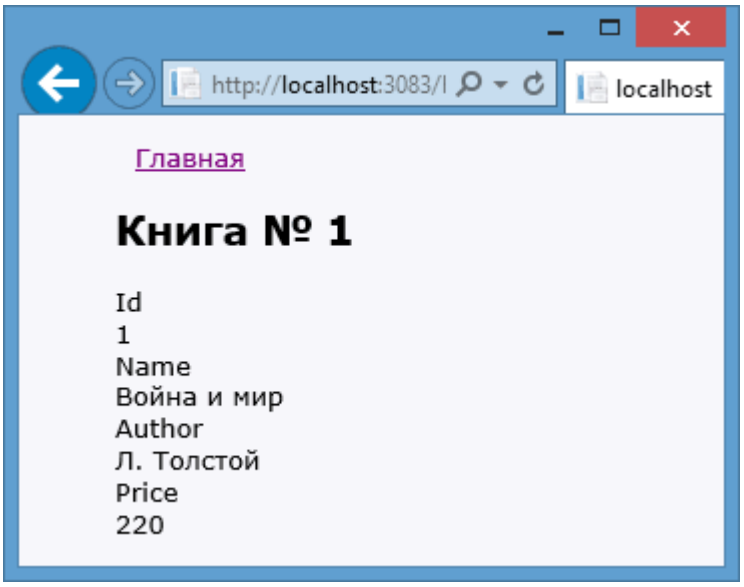
```
public ActionResult BookView(int id)
        {
            Book book = db.Books.Find(id);
            return View(book);
             }
```

и

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

```
@model BookStore.Models.Book
<h2>Книга № @Model.Id</h2>
@Html.DisplayForModel()
```

И обратимся к этому ресурсу, набрав в адресной строке браузера **Home/BookView/1**:



Редактирование модели

В предыдущей теме мы посмотрели, как с помощью шаблонных хелперов отображать значения модели. Теперь же займемся логикой редактирования модели. Вначале добавим в контроллер действие, которые будет получать по **Id** модель и выводить в представление ее свойства для редактирования:

```
[HttpGet]
public ActionResult EditBook(int? id)
        {
            if (id == null)
            {
                return HttpNotFound();
            }
            Book book = db.Books.Find(id);
            if (book != null)
            {
                return View(book);
            }
            return HttpNotFound();
        }
```

На случай, если пользователи не укажут id, мы устанавливаем в качестве параметра не **int**, а **int?**. И если такой параметр не передан, то возвращаем результат метода **HttpNotFound**.

А представление у нас будет содержать набор хелперов **EditorFor** для некоторых полей модели:

```
@{
    ViewBag.Title = "Редактировать книгу";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
@model BookStore.Models.Book
<h2>Книга № @Model.Id</h2>
@using (Html.BeginForm("EditBook", "Home", FormMethod.Post))
{
    <fieldset>
        @Html.HiddenFor(m => m.Id)
        <p>
            @Html.LabelFor(m => m.Name, "Название книги")
            <br />
            @Html.EditorFor(m => m.Name)
        </p>
        <p>
            @Html.LabelFor(m => m.Author, "Автор")
```

```
            <br />
            @Html.EditorFor(m => m.Author)

        </p>
        <p>
            @Html.LabelFor(m => m.Price, "Цена")
            <br />
            @Html.EditorFor(m => m.Price)
        </p>
        <p><input type="submit" value="Отправить" /></p>
    </fieldset>
}
```

Так как уникальный идентификатор **id** книги нам не надо редактировать, то поле для его отображения сделаем скрытым, то есть воспользуемся хелпером **Html.HiddenFor**.

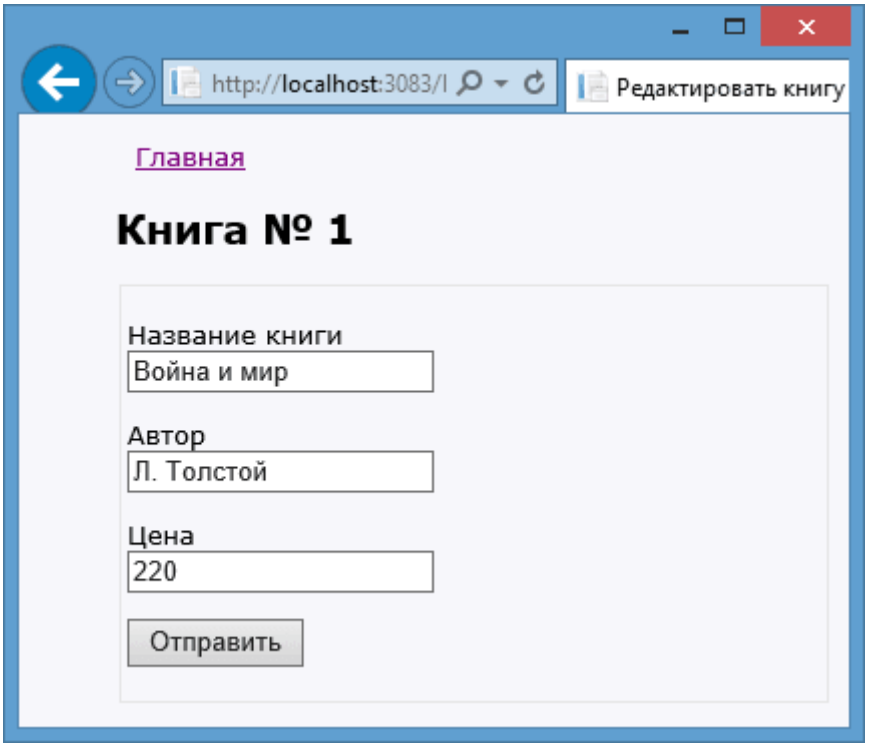
Теперь нам нужен сам код сохранения. Определим в контроллере действие **EditBook**, которое будет обрабатывать **POST**-запросы:

```
[HttpPost]
        public ActionResult EditBook(Book book)
        {
            db.Entry(book).State = EntityState.Modified;
            db.SaveChanges();
            return RedirectToAction("Index");
        }
```

С помощью строки `db.Entry(book).State = EntityState.Modified;` мы указываем, что объект существует **book** уже в базе данных, и для него надо внести в базу измененное значение, а не создавать новую запись. После чего перенаправляемся на главную страницу.

Стоит отметить, что хотя **Entity Framework** позволяет нам абстрагироваться от запросов **sql** и структуры **БД**, но на низком уровне, когда мы устанавливаем значение **db.Entry(book).State = EntityState.Modified;**, то мы тем самым указываем методу **db.SaveChanges()**, что надо сгенерировать и выполнить комаду **UPDATE** для обновления модели в **БД**.

Обратимся к методу **EditBook**, например, **Home/EditBook/1**:



Хелпер **Html.EditorFor** сгенерировал нам поля для редактирования. Мы можем изменить модель, и отправить ее на сервер, где произойдет ее сохранение.

Добавление и удаление модели

Добавление модели

В предыдущей теме мы посмотрели, как редактировать модель. Продолжим работу с моделью **Book** и теперь посмотрим, как мы можем ее добавлять и удалить из **БД**. Для добавления модели вначале определим пару действий:

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

[HttpPost]
public ActionResult Create(Book book)
        {
            db.Books.Add(book);
            db.SaveChanges();
            return RedirectToAction("Index");
        }
```

Первый метод возвращает пользователю представление с формой для добавления, а второй - принимает данные этой формы. Теперь создадим представление. А представление будет выглядеть следующим образом:

```
@model BookStore.Models.Book

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

<h2>Новая книга</h2>

@using (Html.BeginForm())
{
    @Html.LabelFor(model => model.Name, "Название книги")
    <br />
    @Html.EditorFor(model => model.Name)
    <br /><br />
    @Html.LabelFor(model => model.Author, "Автор")
    <br />
    @Html.EditorFor(model => model.Author)
    <br /><br />
    @Html.LabelFor(model => model.Price, "Цена")
    <br />
```

```
    @Html.EditorFor(model => model.Price)
    <br /><br />
    <input type="submit" value="Добавить" />
}
```

При получении модели **book** в действии **Create** метод **db.Books.Add(book)** будет устанавливать значение **Added** в качестве состояния модели. Поэтому метод **db.SaveChanges()** сгенерирует выражение **INSERT** для вставки модели в таблицу. То есть метод **Create** мы могли бы переписать следующим образом:

```
[HttpPost]
public ActionResult Create(Book book)
        {
            db.Entry(book).State = EntityState.Added;
            db.SaveChanges();
            return RedirectToAction("Index");
       }
```

(http://localhost:2044/Home/Create)

Удаление модели

Теперь самая важная часть - удаление модели. Даже не в плане реализации, сколько в плане безопасности. Добавим простое действие, которое удаляет модель из базы данных:

```
public ActionResult Delete(int id)
        {
            Book b = db.Books.Find(id);
            if (b != null)
            {
                db.Books.Remove(b);
                db.SaveChanges();
            }
            return RedirectToAction("Index");
        }
```

Вначале мы проверяем, а есть ли такой объект в **БД**, и если есть, то вызываем метод **db.Books.Remove(b)**. Он установит статус модели в **Deleted**, благодаря чему **Entity Framework** при вызове метода **db.SaveChanges** сгенерирует **sql**-выражение **DELETE**. Но мы можем сами указать статус явным образом:

```
public ActionResult Delete(int id)
        {
            Book b = new Book { Id = id };
            db.Entry(b).State = EntityState.Deleted;
            db.SaveChanges();
            return RedirectToAction("Index");
        }
```

Подобный подход имеет один плюс - мы избегаем первого запроса к бд, который у нас был в выражении **Book b = db.Books.Find(id);**. То есть вместо двух запросов к **БД** теперь у нас только один. Но в целом подобный метод на удаление имеет один минус в плане безопасности.

Допустим, нам пришло электронное письмо, в которое была внедрена картинка посредством тега:

```
<img src="http://адрес_нашего_сайта/Home/Delete/1" />
```

В итоге при открытии письма 1-я запись в таблице может быть удалена. Уязвимость касается не только писем, но может проявляться и в других местах, но смысл один - **GET**-запрос к методу **Delete** несет потенциальную уязвимость. Поэтому переделаем метод следующим образом:

```
[HttpGet]
public ActionResult Delete(int id)
        {
            Book b = db.Books.Find(id);
            if (b == null)
            {
                return HttpNotFound();
            }
            return View(b);
        }

[HttpPost, ActionName("Delete")]
public ActionResult DeleteConfirmed(int id)
        {
            Book b = db.Books.Find(id);
            if (b == null)
            {
                return HttpNotFound();
            }
            db.Books.Remove(b);
            db.SaveChanges();
            return RedirectToAction("Index");
        }
```

Теперь вместо одного метода **Delete** целых два. Атрибут **ActionName("Delete")** указывает, что метод **DeleteConfirmed** будет восприниматься как действие **Delete**. Первый метод передает удаляемую модель в представление. На представлении с помощью нажатия кнопки мы сможем подтвержить удаление. И удаляемый **Id** уйдет второму методу по запросу **POST**. Таким образом, мы уйдем от уязвимости **GET-запроса**. Ну и само представление:

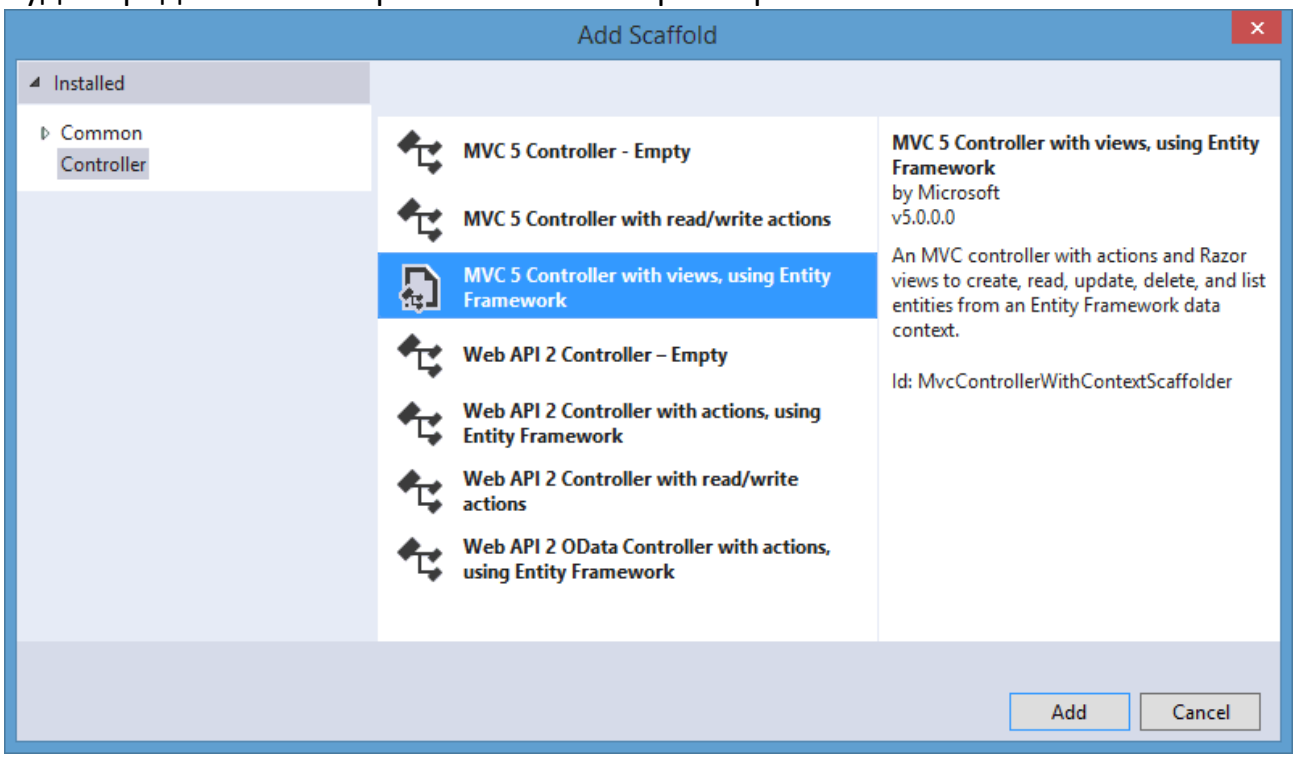
```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}
@model BookStore.Models.Book
<h2>Удаление книги</h2>
<dl>
    <dt>Название</dt>
    <dd>
        @Html.DisplayFor(model => model.Name)
    </dd>

    <dt>Автор</dt>
    <dd>
        @Html.DisplayFor(model => model.Author)
    </dd>

    <dt>Цена</dt>
    <dd>
        @Html.DisplayFor(model => model.Price)
    </dd>
</dl>

@using (Html.BeginForm())
{
    <input type="submit" value="Удалить" />
}
```

Шаблоны формирования

Так как большинство приложений так или иначе основываются на стандартных **CRUD**-операция (**Create - Read - Update - Delete**), то зачастую разработчики вынуждены многократно создавать контроллеры и представления для одних и тех же действий: добавления, изменения, удаления и просмотра записей из **БД**. И чтобы облегчить разработчикам жизнь, команда **MVC** добавила такую полезную функциональность, как **шаблоны**

формирования (**scaffolding templates**). Эти шаблоны позволяют по заданной модели и контексту данных сформировать весь необходимый базовый код контроллеров, а также всю разметку для представлений, с помощью которых можно управлять записями в **БД**.

Чтобы воспользоваться данной функциональностью, добавим новый контроллер. Нажмем правой кнопкой мыши на папку **Controllers** и выберем **Add -> Controller...**. Далее в окне добавления нового контроллера нам будет предложено выбрать шаблон контроллера:



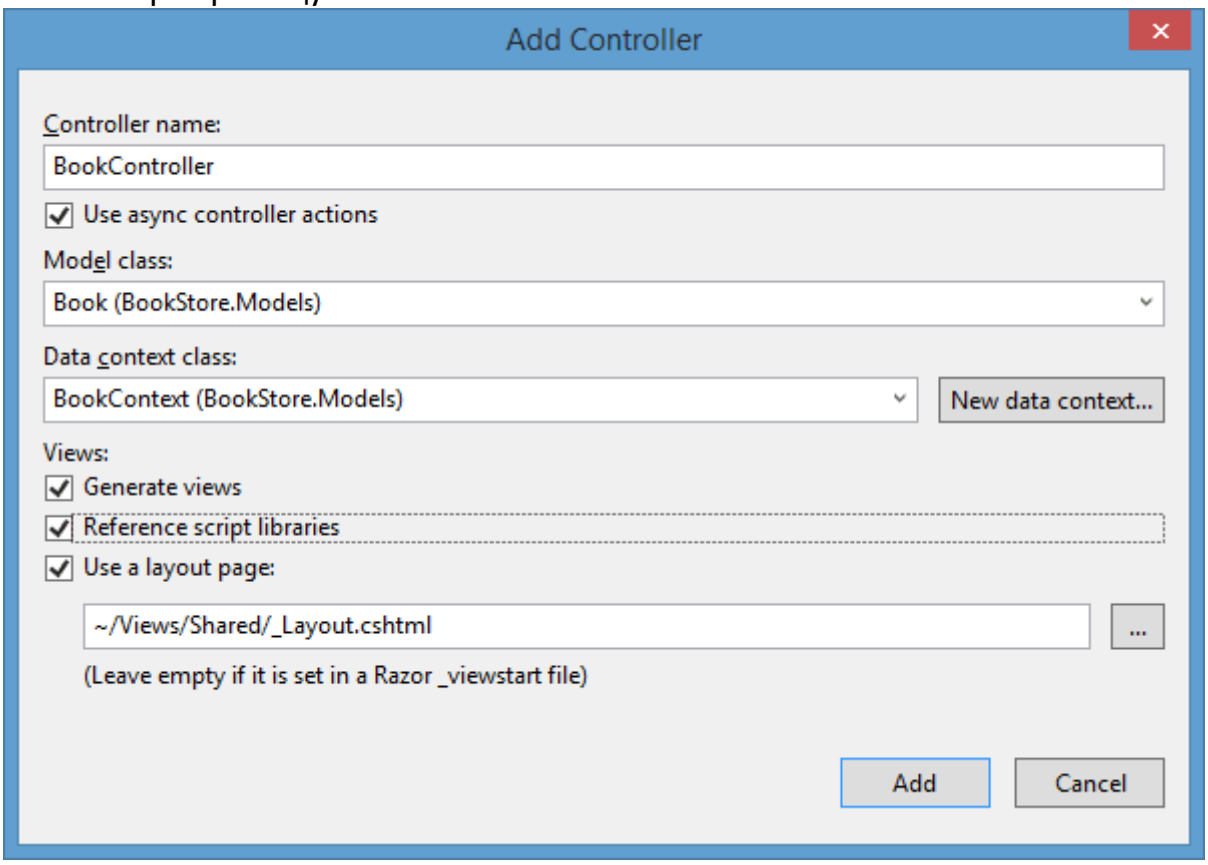
Собственно к **MVC** относятся только первые три шаблона:

- **MVC 5 Controller - Empty**. Этот шаблон добавляет в папку Controllers пустой контроллер, который имеет один единственный метод Index. Данный шаблон не создает представлений
- **MVC 5 Controller with read/write actions**. Данный шаблон добавляет в проект контроллер, который содержит методы Index, Details, Create, Edit и Delete. Однако эти методы не содержат никакой логики работы с базой данных. И нам предлагается самим создать для них код и представления.
- **MVC 5 Controller with views, using Entity Framework**. Это уже более интересный шаблон, который создает контроллер с методами Index, Details, Create, Edit и Delete, а также все необходимые представления для этих действий и добавляет код для извлечения информации из базы данных.

Выберем последний пункт, то есть **MVC 5 Controller with views, using Entity Framework**.

После этого откроется окно добавления нового контроллера, в котором нам будет предложено установить некоторые настройки:

- **Controller name**: имя контроллера;
- **Use async controller actions**: будут ли автоматические сгенерированные методы контроллера асинхронными. Установим данную опцию.
- **Model class**: класс модели. Выберем созданную ранее модель Book (либо какую-то другую имеющуюся модель)
- **Data context class**: класс контекста данных. Выберем контекст данных для выбранной модели.
- **Generate views**: надо ли генерировать представления к создаваемым действиям контроллера. При установке этой опции становятся доступными две следующие опции. Установим все эти опции.
- **Reference script libraries**: будут ли подключать представления библиотеки jquery и другие необходимые файлы javascript
- **Use a layout page**: будут ли генерируемые представления использовать мастер-страницу



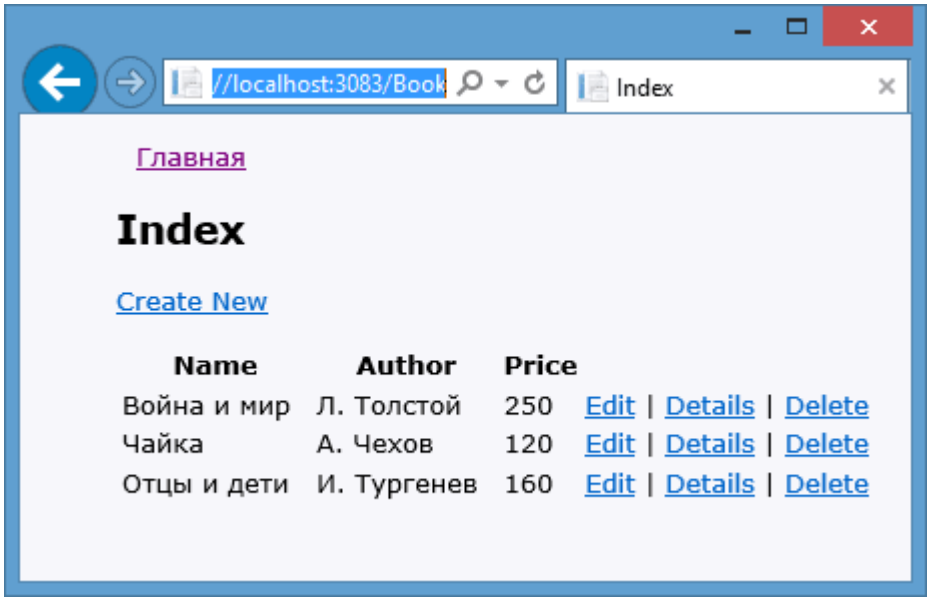
Установив все опции, нажмем кнопку **Add**, и в проект будет добавлен новый контроллер. Он будет выглядеть примерно следующим образом:

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.Entity;
using System.Linq;
using System.Threading.Tasks;
using System.Net;
using System.Web;
using System.Web.Mvc;
using BookStore.Models;

namespace BookStore.Controllers
{
    public class BookController : Controller
    {
        private BookContext db = new BookContext();

        // GET: /Book/
        public async Task<ActionResult> Index()
        {
            return View(await db.Books.ToListAsync());
        }

        // GET: /Book/Details/5
        public async Task<ActionResult> Details(int? id)
        {
            if (id == null)
            {
                return new
HttpStatusCodeResult(HttpStatusCode.BadRequest);
            }
            Book book = await db.Books.FindAsync(id);
            if (book == null)
            {
                return HttpNotFound();
            }
            return View(book);
        }

        // GET: /Book/Create
        public ActionResult Create()
        {
            return View();
        }

        // POST: /Book/Create
```

```
        // To protect from overposting attacks, please enable the
specific properties you want to bind to, for
        // more details see
http://go.microsoft.com/fwlink/?LinkId=317598.
        [HttpPost]
        [ValidateAntiForgeryToken]
        public async Task<ActionResult>
Create([Bind(Include="Id,Name,Author,Price")] Book book)
        {
            if (ModelState.IsValid)
            {
                db.Books.Add(book);
                await db.SaveChangesAsync();
                return RedirectToAction("Index");
            }

            return View(book);
        }

        // GET: /Book/Edit/5
        public async Task<ActionResult> Edit(int? id)
        {
            if (id == null)
            {
                return new
HttpStatusCodeResult(HttpStatusCode.BadRequest);
            }
            Book book = await db.Books.FindAsync(id);
            if (book == null)
            {
                return HttpNotFound();
            }
            return View(book);
        }
        // POST: /Book/Edit/5
        // To protect from overposting attacks, please enable the
specific properties you want to bind to, for
        // more details see
http://go.microsoft.com/fwlink/?LinkId=317598.
        [HttpPost]
        [ValidateAntiForgeryToken]
        public async Task<ActionResult>
Edit([Bind(Include="Id,Name,Author,Price")] Book book)
        {
            if (ModelState.IsValid)
            {
                db.Entry(book).State = EntityState.Modified;
                await db.SaveChangesAsync();
                return RedirectToAction("Index");
```

```
            }
            return View(book);
        }

        // GET: /Book/Delete/5
        public async Task<ActionResult> Delete(int? id)
        {
            if (id == null)
            {
                return new
HttpStatusCodeResult(HttpStatusCode.BadRequest);
            }
            Book book = await db.Books.FindAsync(id);
            if (book == null)
            {
                return HttpNotFound();
            }
            return View(book);
        }

        // POST: /Book/Delete/5
        [HttpPost, ActionName("Delete")]
        [ValidateAntiForgeryToken]
        public async Task<ActionResult> DeleteConfirmed(int id)
        {
            Book book = await db.Books.FindAsync(id);
            db.Books.Remove(book);
            await db.SaveChangesAsync();
            return RedirectToAction("Index");
        }

        protected override void Dispose(bool disposing)
        {
            if (disposing)
            {
                db.Dispose();
            }
            base.Dispose(disposing);
        }
    }
}
```

А в папке **Views/Book** мы найдем все необходимые представления со всем необходимым кодом, который теперь нам не надо набирать вручную. И теперь мы можем запустить проект и перейти в адресной строке браузера к нашему контроллеру (**http://localhost:2044/Book**):



Благодаря шаблонам формирования мы можем не думать о создании кода для стандартных операций, что позволяет сэкономить уйму времени. Правда, после генерации кода все равно придется вносить правки, изменять автоматически сгенерированные названия на свои (например, название страницы, автоматические генерируемых ссылок и др.), однако от основной работы мы уже будем избавлены.

Модели со сложной структурой

Ранее мы использовали относительно простые модели **Book** и **Purchase**. Но в реальных приложениях большинство моделей, как правило, оказываются гораздо сложнее по структуре. Например, создадим две следующие модели, представляющие футболиста и футбольную команду:

```
public class Player
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public int Age { get; set; }
        public string Position { get; set; }
        public int? TeamId { get; set; }
        public Team Team { get; set; }

    }

public class Team
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Coach { get; set; }
        public IEnumerable<Player> Players { get; set; }
    }
```

Кроме стандартных свойств типа string класс **Player** имеет свойство **Team**, которое определяет принадлежность футболиста к определенной команде. Свойство **Team** в данном случае является **навигационным свойством**. Благодаря навигационному свойству мы можем извлекать связанные с объектом данные из **БД**. Но для этого надо также установить внешний ключ.

Внешний ключ состоит из двух свойств: навигационного и обычного. Навигационное мы рассмотрели выше. А обычное должно принимать одно из следующих вариантов имени:

- **Имя_навигационного_свойства+Имя ключа из связанной таблицы** - в нашем случае имя навигационного свойства **Team**, а ключа из модели **Team** - **Id**, поэтому в нашем случае свойство называется **TeamId**.
- **Имя_класса_связанной_таблицы+Имя ключа из связанной таблицы** - в нашем случае класс **Team**, а ключа из модели **Team** - **Id**. И здесь опять же получается TeamId.

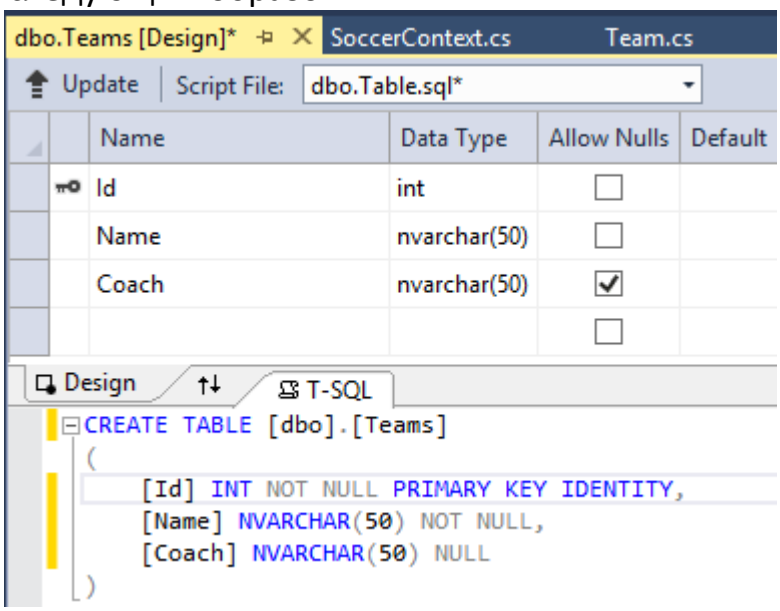
Теперь создадим контекст данных, использующий модели:

```
public class SoccerContext : DbContext
    {
        public DbSet<Player> Players { get; set; }
```

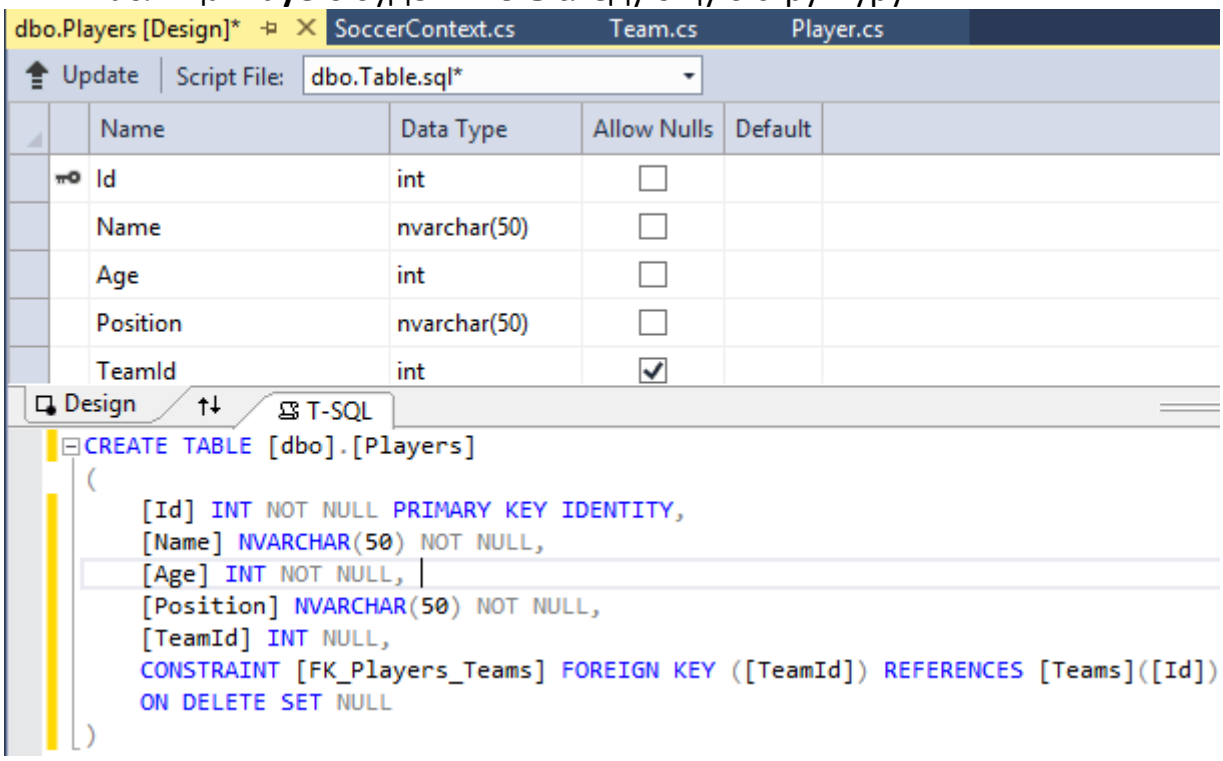
```
        public DbSet<Team> Teams { get; set; }
    }
```

Теперь посмотрим, как бы это все располагалось в **БД**. Например, создадим базу данных **SoccerInfo.mdf**. Пусть для хранения моделей **Player** и **Team** определены соответственно в таблицах **Players** и **Teams**.

Определение таблицы **Teams**, которая будет хранить объекты модели **Team**, выглядит следующим образом:



А таблица **Players** будет иметь следующую структуру:



В отличие от таблицы **Teams** здесь мы также задаем внешний ключ - свойство **TeamId** теперь будет ссылаться на поле **Id** из таблицы **Teams**.

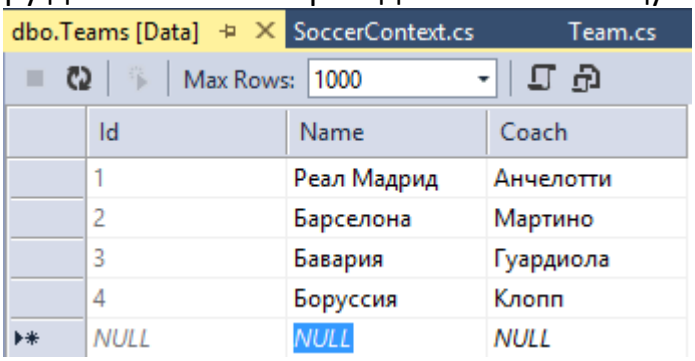
Чтобы задать внешний ключ, мы добавляем в панели **SQL** внизу под дизайнером таблицы следующую строку:

```
CONSTRAINT [FK_Players_Teams] FOREIGN KEY ([TeamId]) references
    [Teams]([Id])
```

Это обычное выражение языка **SQL**, которое связывает столбцы двух таблиц. Последняя часть этого выражения (`ON DELETE SET NULL`) указывает, что при удалении объекта из таблицы **Teams**, свойству **TeamId**, которое ссылалось на удаленный объект, будет присвоено значение **`null`**.

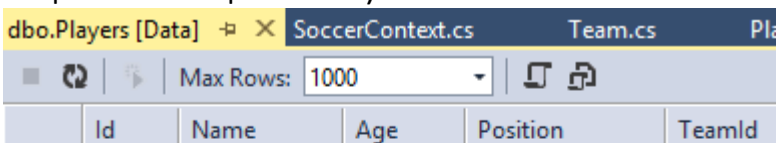
Это надо, чтобы у нас игроки при удалении команд не относились больше к удаленным командам. Однако мы можем задать и другое действие, например, при удалении команды удалить всех ее игроков. Для этого нам надо написать **ON DELETE CASCADE**.

Теперь после определения таблиц наполним их некоторыми начальными данными. К примеру добавим некоторые данные в таблицу **Teams**:



Id	Name	Coach
1	Реал Мадрид	Анчелотти
2	Барселона	Мартино
3	Бавария	Гуардиола
4	Боруссия	Клопп
NULL	NULL	NULL

И в таблицу **Players** (где столбец **TeamId** содержит некоторое существующее значение из столбца Id таблицы Teams):



Id	Name	Age	Position	TeamId
1	Месси	26	Нападающий	2
2	Роналду	29	Нападающий	1
3	Бейл	24	Полузащитник	1
4	Неймар	22	Нападающий	2
5	Иньеста	29	Полузащитник	2
6	Рибери	30	Полузащитник	3
NULL	NULL	NULL	NULL	NULL

Теперь перейдем к созданию логики приложения. Добавим в приложение контроллер и определим в нем вывод всех игроков на страницу:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using NavigationProperty.Models;
using System.Data.Entity;


namespace NavigationProperty.Controllers
{
    public class SoccerController : Controller
    {
        SoccerContext db = new SoccerContext();
        // Выводим всех футболистов
        public ActionResult Index()
        {
            var players = db.Players.Include(p => p.Team);
            return View(players.ToList());
        }

     }
}
```

Теперь с помощью метода **Include** фреймворк подгружает для каждого игрока команду, связанную с данным игроком через внешний ключ. И создадим представление **Index.cshtml**, которое будет выводить всех игроков:

```
@model IEnumerable<NavigationProperty.Models.Player>
@{
    ViewBag.Title = "Каталог игроков";
}

<h2>Каталог игроков</h2>
<p>
    @Html.ActionLink("Добавить игрока", "Create")
</p>
<table>
    <tr>
        <th>Имя игрока</th>
        <th>Возраст</th>
        <th>Позиция на поле</th>
        <th>Команда</th>
        <th></th>
    </tr>
```

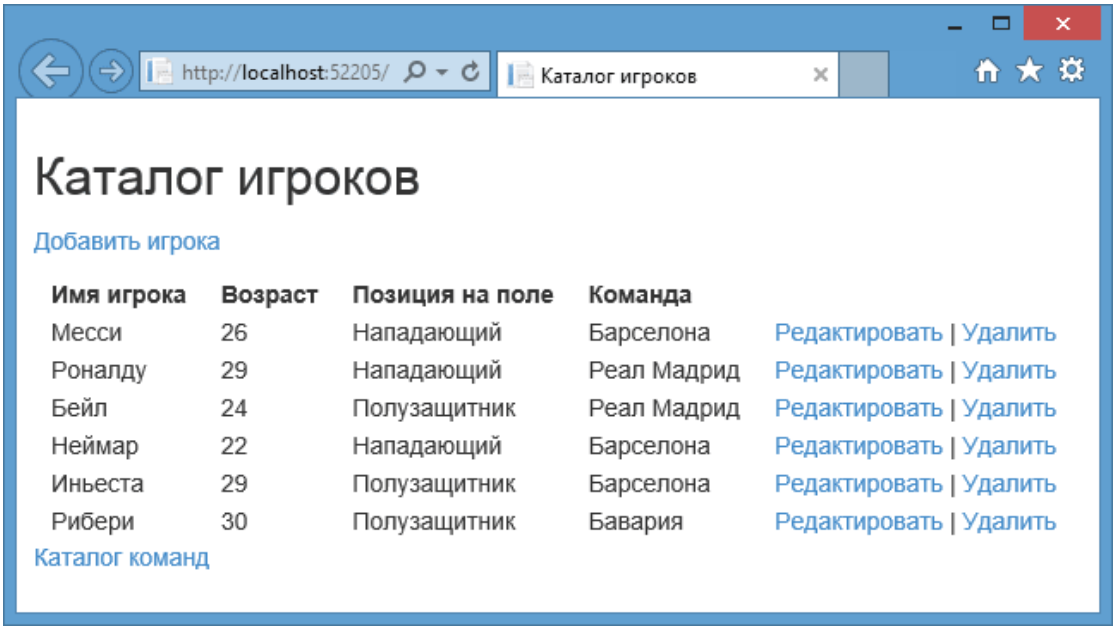
```
    @foreach (var item in Model)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Name)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Age)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Position)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Team.Name)
            </td>
            <td>
                @Html.ActionLink("Редактировать", "Edit", new { id =
item.Id }) |
                @Html.ActionLink("Удалить", "Delete", new { id = item.Id
})
            </td>
        </tr>
    }
</table>
<p>
    @Html.ActionLink("Каталог команд", "ListTeams")
</p>
```

и в конце надо добавит в файле **web.config**:

```
<connectionStrings>
    <add name="SoccerContext" connectionString="Data
Source=(LocalDB)\v11.0;AttachDbFilename='|DataDirectory|\SoccerInfo.mdf';
Integrated Security=True"
   providerName="System.Data.SqlClient" />
  </connectionStrings>
```

(**http://localhost:2044/Soccer/Index**)

Так как в методе контроллера с помощью метода **Include** ко всем моделям **Player** подключается свой объект **Team** по навигационному свойству **TeamId**, то то в представлении мы можем получить этот связанный объект **Team** и использовать его свойства, например, получить **item.Team.Name** для отображения имени команды.



Подобным образом можно вывести список команд. Но там все просто и не так интересно. Зато у нас в модели **Team** свойство **Players**, которое призвано хранить связанных с командой игроков. Используем его. Например, выведем все данные о команде, в том числе о ее игроках. Вначале добавим в контроллер следующий метод:

```
public ActionResult TeamDetails(int? id)
        {
            if (id == null)
            {
                return HttpNotFound();
            }
            Team team = db.Teams.Find(id);
            if (team == null)
            {
                return HttpNotFound();
            }
            team.Players = db.Players.Where(m => m.TeamId == team.Id);
            return View(team);
        }
```

Во-первых, чтобы обработать ввод при отсутствии передаваемого значения, в качестве параметра используем **int? id**. Во-вторых, мы подгружаем всех игроков, связанных с командой, в выражении

```
team.Players = db.Players.Where(m => m.TeamId == team.Id);
```

Ну и представление **TeamDetails.cshtml** для отображения данных о команде могло бы выглядеть так:

```
@using NavigationProperty.Models
@model Team

@{
    ViewBag.Title = "Команда " + @Model.Name;
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<div>
    <h4>Команда @Model.Name</h4>
    <hr />
    <dl>
        <dt>Название</dt>

        <dd>
            @Html.DisplayFor(model => model.Name)
        </dd>

        <dt>Тренер</dt>

        <dd>
            @Html.DisplayFor(model => model.Coach)
        </dd>

        <dt>Игроки</dt>

        <dd>
            <ul>
                @foreach (Player player in Model.Players)
                {
                    <li>@player.Name (@player.Position)</li>
                }
            </ul>
        </dd>
    </dl>
</div>
```

(http://localhost:34362/Home/TeamDetails/1)

Работа со сложными моделями

В предыдущей теме мы создали две модели **Player** и **Team** и вывели элементы из таблицы **Players** на страницу. Теперь посмотрим, как проделать остальные операции с моделями. В общем-то тут будет все то же самое, что и с обычными моделями. Единственное отличие - мы должны учитывать значения навигационное свойство, имеющееся в сложной модели.

Добавление модели

При добавлении модели, имеющий внешний ключ, характерно все то же самое, что и для обычной модели. Единственное дополнение - нам надо также передавать в представление набор значений для связи внешнего ключа с другой таблицей. Итак, добавим в контроллер следующее действие **Create**:

```
[HttpGet]
public ActionResult Create()
        {
            // Формируем список команд для передачи в представление
            SelectList teams = new SelectList(db.Teams, "Id", "Name");
            ViewBag.Teams = teams;
            return View();
        }

[HttpPost]
public ActionResult Create(Player player)
        {
            //Добавляем игрока в таблицу
            db.Players.Add(player);
            db.SaveChanges();
            // перенаправляем на главную страницу
            return RedirectToAction("Index");
        }
```

Первый метод обрабатывает GET-запрос и возвращает представление, передавая в него объект **SelectList** - список всех команд.

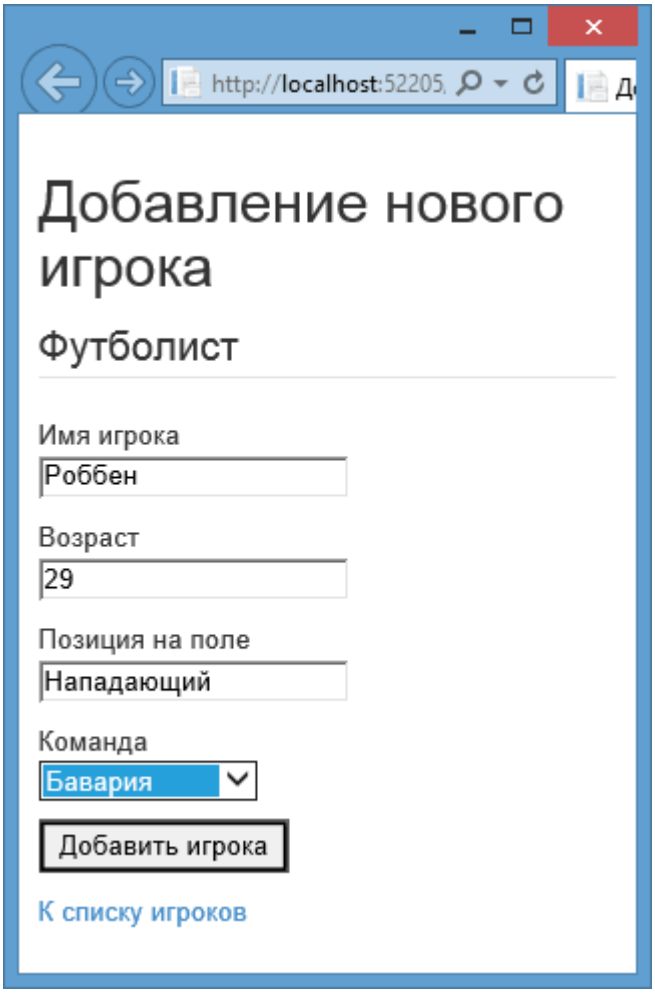
Второй метод получает введенную пользователем в представлении модель и добавляет ее в **БД**. А теперь создадим представление **Create.cshtml**:

```
@model NavigationProperty.Models.Player

@{
    ViewBag.Title = "Добавление игрока";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
```

```
<h2>Добавление нового игрока</h2>

@using (Html.BeginForm())
{
    <fieldset>
        <legend>Футболист</legend>

        <p>
            Имя игрока <br />
            @Html.EditorFor(model => model.Name)
        </p>

        <p>
            Возраст <br />
            @Html.EditorFor(model => model.Age)
        </p>

        <p>
            Позиция на поле  <br />
            @Html.EditorFor(model => model.Position)
        </p>
        <p>
            Команда <br />
            @Html.DropDownListFor(model => model.TeamId, ViewBag.Teams as
SelectList)
        </p>

        <p>
            <input type="submit" value="Добавить игрока" />
        </p>
    </fieldset>
}
<div>
    @Html.ActionLink("К списку игроков", "Index")
</div>
```

Как и в случае с простыми моделями, мы привязываем поля к определенному свойству.

Тут следует лишь отметить создание выпадающего списка, из которого мы выбираем команду. Выбираемое значение в этом списке привязывается к свойству модели **TeamId**.



Редактирование модели

Редактирование работает подобным способом. Определим в контроллере следующее действие **Edit**:

```
[HttpGet]
public ActionResult Edit(int? id)
        {
            if (id == null)
            {
                return HttpNotFound();
            }
            // Находим в бд футболиста
            Player player = db.Players.Find(id);
            if (player != null)
            {
                // Создаем список команд для передачи в представление
                SelectList teams = new SelectList(db.Teams, "Id", "Name",
player.TeamId);
                ViewBag.Teams = teams;
                return View(player);
```

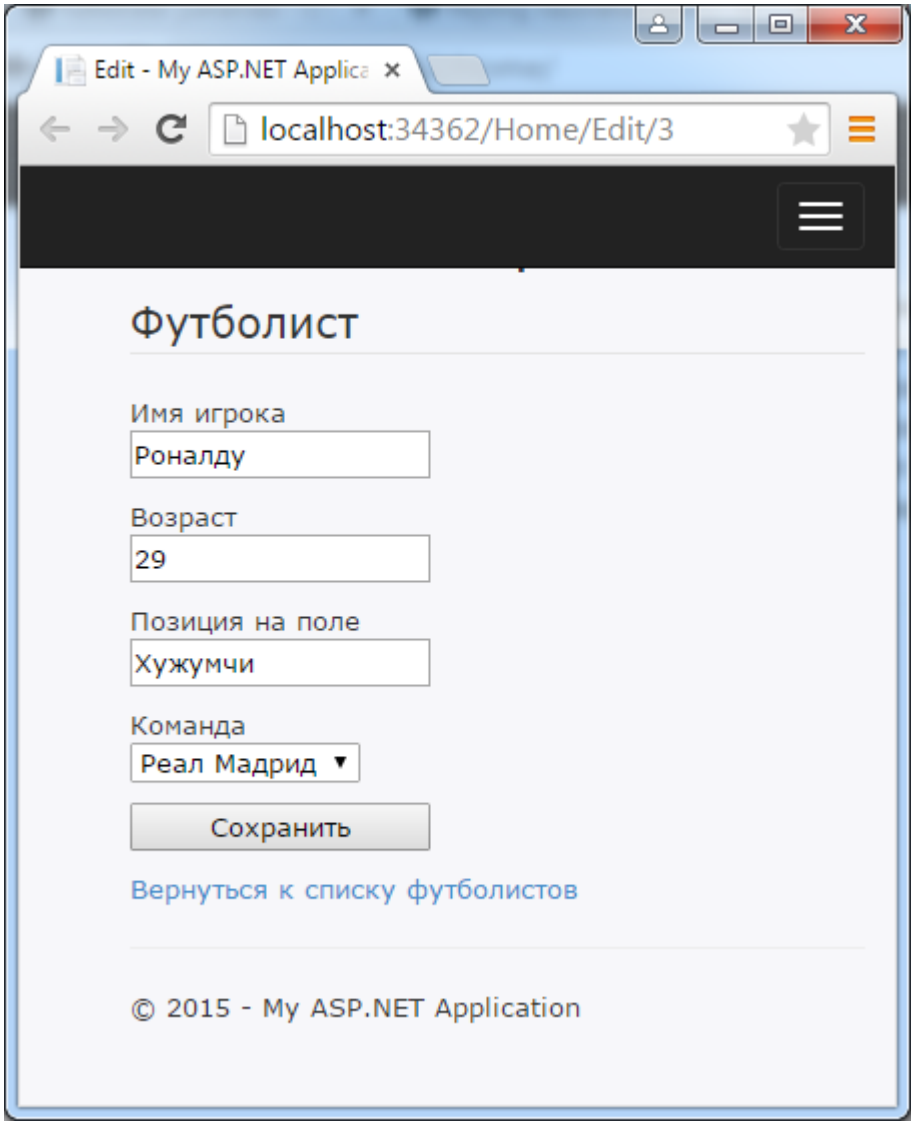
```
            }
            return RedirectToAction("Index");
        }

[HttpPost]
public ActionResult Edit(Player player)
        {
            db.Entry(player).State = EntityState.Modified;
            db.SaveChanges();
            return RedirectToAction("Index");
        }
```

Здесь также в виде объекта **SelectList** создается список команд, которые извлекаются из **БД**. И после получения запроса на редактирования определенной модели **Player** контроллер передает эту модель и список команд в представление **Edit.cshtml**:

```
@model NavigationProperty.Models.Player
@{
    ViewBag.Title = "Edit";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<h2>Изменение игрока</h2>

@using (Html.BeginForm())
{
    <fieldset>
        <legend>Футболист</legend>

        @Html.HiddenFor(model => model.Id)

        <p>
            Имя игрока <br />
            @Html.EditorFor(model => model.Name)
        </p>

        <p>
            Возраст <br />
            @Html.EditorFor(model => model.Age)
        </p>

        <p>
            Позиция на поле  <br />
            @Html.EditorFor(model => model.Position)
        </p>
        <p>
```

```
            Команда <br />
            @Html.DropDownListFor(model => model.TeamId, ViewBag.Teams as
SelectList)
        </p>
        <p>
            <input type="submit" value="Сохранить" />
        </p>
    </fieldset>
}
<div>
    @Html.ActionLink("Вернуться к списку футболистов", "Index")
</div>
```

http://localhost:34362/Home/Edit/3



Ну и удаление производится также, как и в случае с обычной моделью.

```
[HttpGet]
public ActionResult Delete(int id)
```

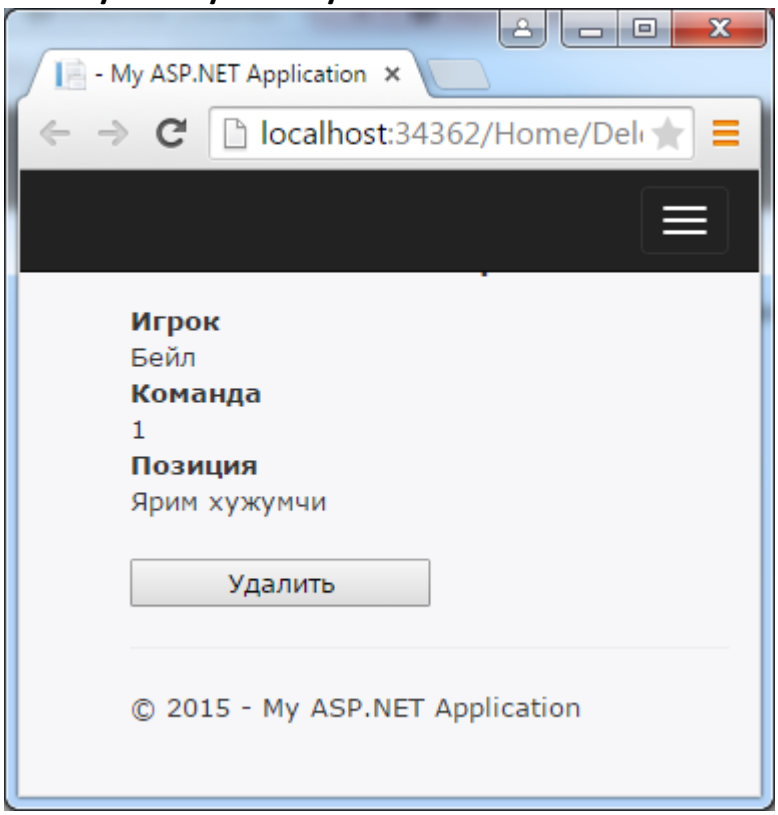
```
        {
            if (id == null)
            {
                return HttpNotFound();
            }

            // Находим в бд футболиста
            Player player = db.Players.Find(id);
            if (player != null)
            {
                //// Создаем список команд для передачи в представление
                //SelectList teams = new SelectList(db.Teams, "Id",
"Name", player.TeamId);
                //ViewBag.Teams = teams;
                return View(player);
            }
            return RedirectToAction("Index");
        }

[HttpPost, ActionName("Delete")]
public ActionResult DeleteConfirmed(int id)
        {
            Player b = db.Players.Find(id);
            if (b == null)
            {
                return HttpNotFound();
            }
            db.Players.Remove(b);
            db.SaveChanges();
            return RedirectToAction("Index");
        }
```

И после получения запроса на удаление определенной модели **Player** контроллер передает эту модель в представление **Delete.cshtml**:

```
@model NavigationProperty.Models.Player
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<h2>Удаление игрока</h2>
<dl>
    <dt>Игрок</dt>
    <dd>
        @Html.DisplayFor(model => model.Name)
    </dd>

    <dt>Команда</dt>
```

```
    <dd>
        @Html.DisplayFor(model => model.TeamId)
    </dd>

    <dt>Позиция</dt>
    <dd>
        @Html.DisplayFor(model => model.Position)
    </dd>
</dl>

@using (Html.BeginForm())
{
    <input type="submit" value="Удалить" />
}
```

http://localhost:34362/Home/Delete/6



Модели со связью многие-ко-многим

Кроме моделей со связью один-ко-одному и один-ко-многим, которые были рассмотрены в прошлых темах, модели по типу "многие-ко-многим". Если посмотреть на мир вокруг себя, то мы сможем найти подобные модели. Наиболее распространенный и хрестоматийный пример - учеба в университете, где различное количество студентов может посещать различное количество

дисциплин. И при этом у нас может возникнуть необходимость, как вести учет студентов по конкретной дисциплине, так и вести учет различных дисциплин для конкретного студента. Попробуем смоделировать данную ситуацию в приложении **ASP.NET MVC 5**.

Во-первых, создадим новый проект. Первым делом нам надо создать модели. Сразу скажу, что для простоты будем использовать подход **Code First**.

Итак, у нас есть две модели - студент и курс университетской дисциплины. Сначала добавим модель **Student** со следующим содержанием:

```
public class Student
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Surname { get; set; }
        public virtual ICollection<Course> Courses { get; set; }

    }
```

И также добавим модель **Course**:

```
public class Course
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public virtual ICollection<Student> Students { get; set; }

    }
```

Модели довольно простые за исключением виртуальных свойств - **Students** и **Courses** - благодаря этим свойствам и будет происходить связь многие-ко-многим.

Следующий этап - создание контекста данных. Добавим в проект следующий класс **StudentsContext**:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data.Entity;

namespace Study.Models
{
    public class StudentsContext : DbContext
    {
        public DbSet<Student> Students { get; set; }
        public DbSet<Course> Courses { get; set; }
```

```
        public StudentsContext()
            : base("DefaultConnection")
        { }

        protected override void OnModelCreating(DbModelBuilder
modelBuilder)
        {
            modelBuilder.Entity<Course>().HasMany(c => c.Students)
                .WithMany(s => s.Courses)
                .Map(t => t.MapLeftKey("CourseId")
                .MapRightKey("StudentId")
                .ToTable("CourseStudent"));
        }
    }

}
```

Во-первых, чтобы использовать строку подключения по умолчанию, установим для нее контекст данных в конструкторе:

```
public StudentsContext() : base("DefaultConnection") { }
```

Дальше идет самое интересное. В создаваемой базе данных все данные о студентах будут храниться в таблице **Students**, а данные о университетских курсах - в таблице **Courses**. Но эти таблицы должны быть как-то связаны связью многие-ко-многим. И эту связь обеспечит еще одна таблица, которая будет называться **CourseStudent**.

Для построения этой таблицы мы переопределяем метод **OnModelCreating**, в котором с помощью объекта **modelBuilder** создаем новую таблицу и определяем ее поля. Одно ее поле - **CourseId** - будет ссылаться на таблицу **Courses** и хранить в себе **Id** курса. А второе поле - **StudentId** - будет ссылаться на таблицу студентов и хранить **Id** студента.

В итоге у нас получится набор пар **Id** курса - **Id** студента, благодаря этому мы сможем определить связь многие-ко-многим. И теперь проинициализируем базу данных начальными данными. Добавим в проект следующий класс:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Data.Entity;

namespace Study.Models
{
```

```
    public class CourseDbInitializer :
DropCreateDatabaseAlways<StudentsContext>
    {
        protected override void Seed(StudentsContext context)
        {
            Student s1 = new Student { Id = 1, Name = "Егор", Surname =
"Иванов" };
            Student s2 = new Student { Id = 2, Name = "Мария", Surname =
"Васильева" };
            Student s3 = new Student { Id = 3, Name = "Олег", Surname =
"Кузнецов" };
            Student s4 = new Student { Id = 4, Name = "Ольга", Surname =
"Петрова" };

            context.Students.Add(s1);
            context.Students.Add(s2);
            context.Students.Add(s3);
            context.Students.Add(s4);

            Course c1 = new Course
            {
                Id = 1,
                Name = "Операционные системы",
                Students = new List<Student>() { s1, s2, s3 }
            };
            Course c2 = new Course
            {
                Id = 2,
                Name = "Алгоритмы и структуры данных",
                Students = new List<Student>() { s2, s4 }
            };
            Course c3 = new Course
            {
                Id = 3,
                Name = "Основы HTML и CSS",
                Students = new List<Student>() { s3, s4, s1 }
            };

            context.Courses.Add(c1);
            context.Courses.Add(c2);
            context.Courses.Add(c3);

            base.Seed(context);
        }
    }

}
```

Обратите внимание, как обеспечивается связь между курсами и студентами: мы просто добавляем набор созданных студентов в коллекцию **Students** для каждого курса. И чтобы все это заработало, добавим в файл **Global.asax.cs** в метод **Application_Start** следующую строчку:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Optimization;
using System.Web.Routing;
using System.Data;
using System.Data.Entity;
using Study.Models;

namespace Study
{
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            Database.SetInitializer(new CourseDbInitializer());

            AreaRegistration.RegisterAllAreas();
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);
        }
    }
}
```

Создадим контроллер. Так как вывод просто таблицы как курсов, так студентов не представляет сложности, то мы его разибрать не будем. Нас интересует получение связанных данных. Поэтому добавим контроллер **HomeController** и определим в нем следующий метод **Details**, который будет выводить информацию по конкретному студенту:

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.Entity;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using Study.Models;
```

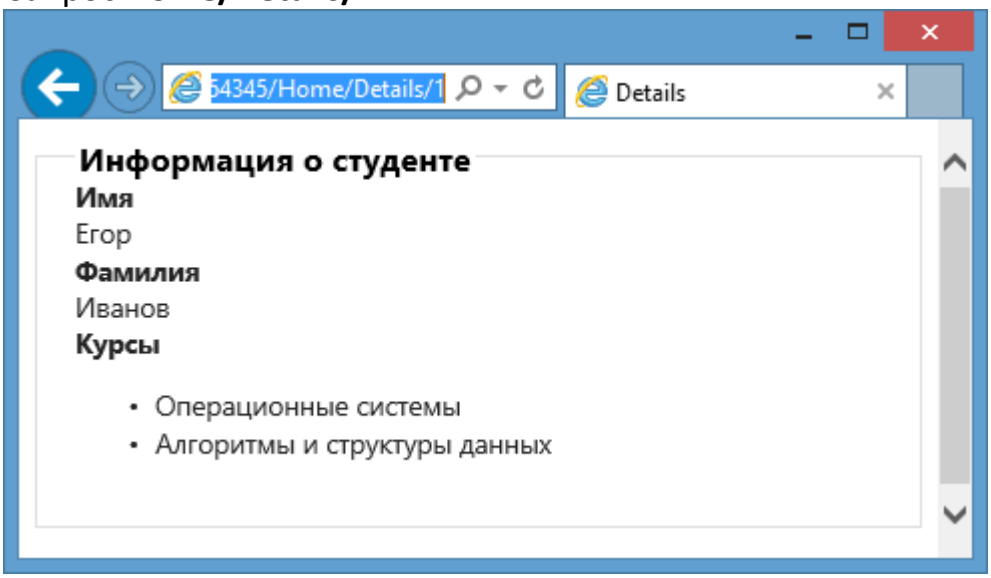
```
namespace Study.Controllers
{
    public class HomeController : Controller
    {
        private StudentsContext db = new StudentsContext();
        public ActionResult Index()
        {
            return View(db.Students.ToList());
        }

        public ActionResult Details(int id = 0)
        {
            Student student = db.Students.Find(id);
            if (student == null)
            {
                return HttpNotFound();
            }
            return View(student);
        }

        protected override void Dispose(bool disposing)
        {
            db.Dispose();
            base.Dispose(disposing);
        }
    }
}
```

Метод **Details** представляет обычный метод получения информации по объекту **Student**. И благодаря определению в модели **Student** виртуального свойства **Courses**, все связанные с данным студентом университетские курсы будут автоматически подключаться. Теперь добавим представление **Details.cshtml**:

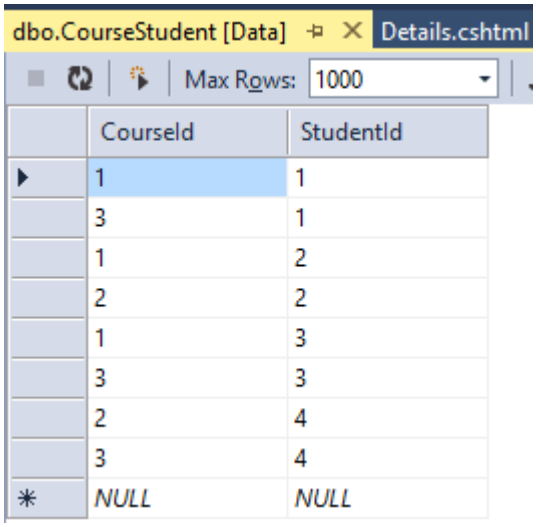
```
@using Study.Models
@model Student
@{
    ViewBag.Title = "Details";
}

<fieldset>
    <legend>Информация о студенте</legend>

    <div class="display-label"><b>Имя</b></div>
    <div class="display-field">
        @Html.DisplayFor(model => model.Name)
```

```
    </div>

    <div class="display-label"><b>Фамилия</b></div>
    <div class="display-field">
        @Html.DisplayFor(model => model.Surname)
    </div>

    <div class="display-label"><b>Курсы</b></div>
    <ul>
        @foreach (Course c in Model.Courses)
        {
            <li>@c.Name</li>
        }
    </ul>
</fieldset>
```

Поскольку к модели автоматически цепляются связаные данные, то мы их можем получить через **Model.Courses**. Так как у нас в базе данных уже есть начальные данные, то мы можем получить информацию по первому студенту, отправив запрос **Home/Details/1**:



И также если мы посмотрим на базу данных, то обнаружим в ней, как и ожидалось, три таблицы. Связующая таблица **CourseStudent** будет иметь следующее содержание:


dbo.CourseStudent [Data] Details.cshtml

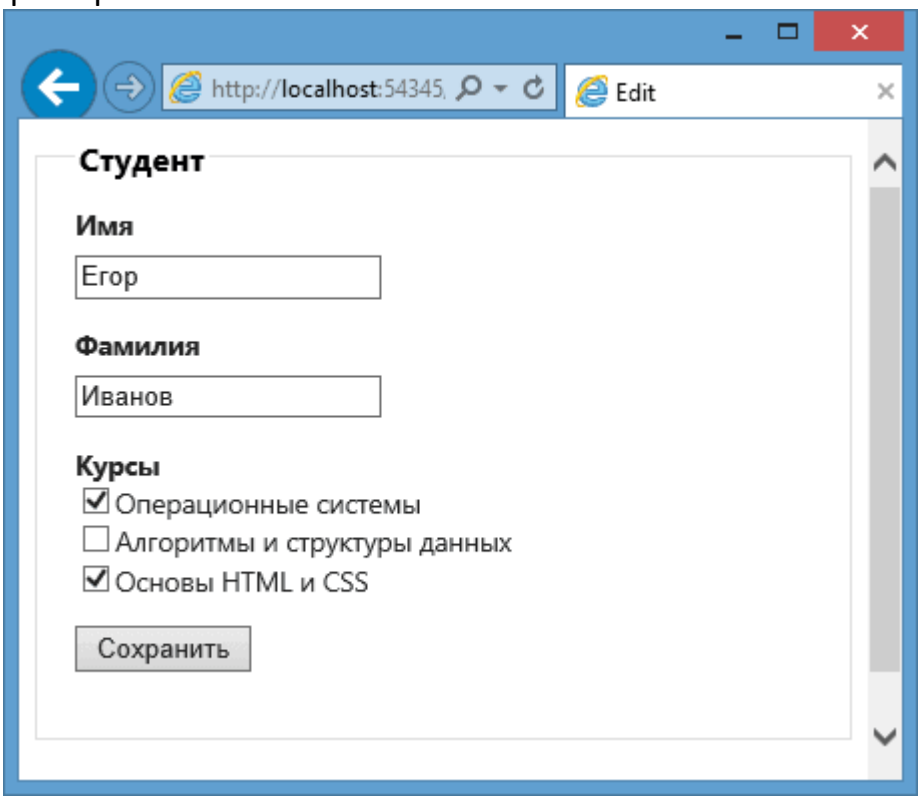
Max Rows: 1000

CourseId	StudentId
1	1
3	1
1	2
2	2
1	3
3	3
2	4
3	4
NULL	NULL

То есть все те же данные, которые мы указали для начальной инициализации базы данных.

Работа с моделями со связью многие-ко-многим

Простой вывод связанных данных со связью "многие-ко-многим" не представляет особого труда. Однако не совсем понятно, как делать создание и редактирование подобных моделей. Продолжим начатый в прошлой теме пример со студентами и курсами и внесем в него функциональность редактирования студентов. В итоге форма редактирования у нас будет выглядеть примерно так:



Итак, добавим в контроллер **HomeController** следующее действие **Edit**:

```
public ActionResult Edit(int id = 0)
        {
            Student student = db.Students.Find(id);
            if (student == null)
            {
                return HttpNotFound();
            }
            ViewBag.Courses = db.Courses.ToList();
            return View(student);
        }

[HttpPost]
public ActionResult Edit(Student student, int[] selectedCourses)
        {
            Student newStudent = db.Students.Find(student.Id);
            newStudent.Name = student.Name;
```

```
            newStudent.Surname = student.Surname;

            newStudent.Courses.Clear();
            if (selectedCourses != null)
            {
                //получаем выбранные курсы
                foreach (var c in db.Courses.Where(co =>
selectedCourses.Contains(co.Id)))
                {
                    newStudent.Courses.Add(c);
                }
            }

            db.Entry(newStudent).State = EntityState.Modified;
            db.SaveChanges();
            return RedirectToAction("Index");
        }
```

Действие **Edit** представляет два метода - для запроса **GET** и для запроса **POST**. Метод, обрабатывающий запрос **GET**, стандартный - он передает в представление редактируемую модель, а также список всех курсов через
`ViewBag.Courses = db.Courses.ToList();`
чтобы мы могли затем эти курсы вывести в представлении.

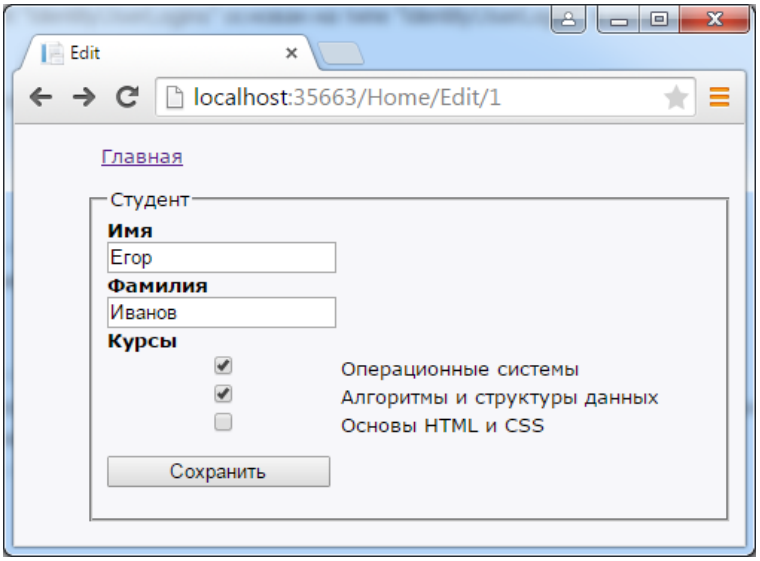
Метод **POST** принимает полученные данные, только кроме модели **Student** сюда также передаются все выбранные курсы в виде массива **Id** курсов. В этот массив и будут помещаться все значения всех отмеченных на форме флажков. В самом методе после прохождения валидации мы устанавливаем новые значения свойств модели.

Затем нам надо установить в коллекции **Courses** у студента все отмеченные курсы. Для этого проходим по всем курсам из базы данных, и если они были отмечены на форме и отсутствуют в списке, то добавляем их. Неотмеченные удаляем (если они есть в списке). В итоге информация в базе данных будет соотвествующим образом обновлена.
И напоследок добавим само представление **Edit.cshtml**:

```
@using Study.Models
@model Student
@{
    ViewBag.Title = "Edit";
}

@using (Html.BeginForm())
{
    <fieldset>
        <legend>Студент</legend>
```

```
        @Html.HiddenFor(model => model.Id)

        <div class="editor-label"><b>Имя</b></div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Name)
        </div>

        <div class="editor-label"><b>Фамилия</b></div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Surname)
        </div>
        <div class="editor-label"><b>Курсы</b></div>
        @{
    List<Course> courses = ViewBag.Courses;
    foreach (Course c in courses)
    {
        <input type="checkbox" name="selectedCourses" value="@c.Id"
               @(Model.Courses.Contains(c) ? "checked=\"checked\"" : "")
/>@c.Name <br />
    }
        }

        <p>
            <input type="submit" value="Сохранить" />
        </p>
    </fieldset>
}
```

Ну а сохранение модели будет во многом идентично редактированию.
http://localhost:35663/Home/Edit/1



Передача массивов и сложных данных в контроллер

Из предыдущих тем мы узнали, как передавать отдельные объекты из представления в методы действия контроллера в качестве параметров. Но в реальности может возникнуть ситуация, что потребуется передать в метод не один объект типа **int** или какой-нибудь модели, а сразу несколько объектов. Посмотрим на некоторые возможные случаи.

Передача коллекции

Определим следующую форму в представлении:

```
@using (Html.BeginForm())
{
    @Html.TextBox("names")
    @Html.TextBox("names")
    @Html.TextBox("names")
    @Html.TextBox("names")
    <input type="submit" />
}
```

Таким образом, у нас в **html-разметке** будет создано четыре элемента **input**

```
<form action="/Home/Array" method="post">
    <input id="names" name="names" type="text" value="" />
    <input id="names" name="names" type="text" value="" />
    <input id="names" name="names" type="text" value="" />
    <input id="names" name="names" type="text" value="" />
    <input type="submit" />
</form>
```

Поэтому при отправке формы будет формироваться коллекция из **names**, состоящая из четырех элементов. И в методе контроллера мы сможем получить все эти элементы:

```
[HttpPost]
public string Array(List<string> names)
        {
            string fin = "";
            for (int i = 0; i < names.Count; i++)
            {
                fin += names[i] + ";  ";
            }
            return fin;
        }
```

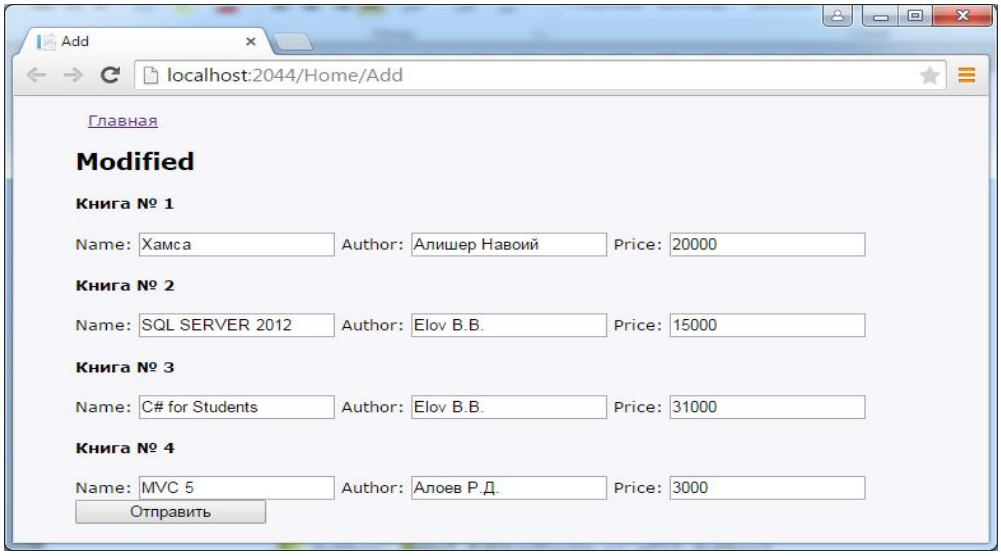
Передача коллекции объектов модели

Мы можем передать в представление массив объектов некоторой модели, например, модели **Book**:

```
[HttpGet]
        public ActionResult Add()
        {
            return View(db.Books.ToList());
        }
```

Чтобы вывести объекты для редактирования в представление мы можем использовать следующую конструкцию: **http://localhost:2044/Home/Add**

```
@model List<BookStore.Models.Book>
@{
    ViewBag.Title = "Add";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<h2>Modified</h2>

@using (Html.BeginForm())
{
    for (int i = 0; i < Model.Count; i++)
    {
        <h4>Книга № @(i + 1)</h4>

        @: Name: @Html.EditorFor(m => m[i].Name)
        @: Author: @Html.EditorFor(m => m[i].Author)
        @: Price: @Html.EditorFor(m => m[i].Price)
    }
    <input type="submit" />
}
```



Так мы сгенерируем для каждого объекта набор полей для редактирования его свойств. И после нажатия кнопки весь данный массив отправится на сервер, где его можно получить таким образом:

```
[HttpPost]
        public string Add(List<Book> books)
        {
            //.......................
       }
```

Передача разных объектов одной модели

В предыдущем случае мы передавали коллекцию объектов модели **Book**. Однако может возникнуть ситуация, когда мы должны разграничить как-то переданные объекты, а не рассматривать их как одну коллекцию. Например, метод контроллера может выглядеть так:

```
[HttpPost]
        public string Add(Book book, Book myBook)
        {
            //.......................
        }
```

Мы используем два отдельных объекта **Book**. Как мы можем передать их из представления в контроллер? Допустим, теперь, что один объект мы передаем в качестве модели представления, а другой создаем в том же представлении:

```
@using BookStore.Models
@model Book
@{
    ViewBag.Title = "Array";
```

```
    Layout = "~/Views/Shared/_Layout.cshtml";
}
@{
    Book myBook = new Book() { Name = "Мартин Иден", Author = "Джек
Лондон", Price = 190 };
}
<h2>Книги</h2>

@using (Html.BeginForm())
{
    @Html.EditorFor(m => myBook)
    <p></p>
    @Html.EditorForModel()
    <input type="submit" />
}
```

Одну модель мы передаем из контроллера в представление:

```
[HttpGet]
public ActionResult Array()
        {
            Book firstBook = db.Books.ToList<Book>().FirstOrDefault();
            return View(firstBook);
        }
```

Вторую модель - **myBook** мы создаем уже в самом представлении. Все поля модели генерируются с помощью хелпера **@Html.EditorFor(m=>myBook)**.

Обратите внимание на имя модели - **myBook**. Так как мы ожидаем, что данный объект должен быть передан в метод в качестве параметра **myBook**, то он должен иметь точно такое же имя, а не произвольное.

Таким образом, мы можем передать на сервер два разных объекта одной модели.

http://localhost:2044/Home/Array

Передача сложных объектов

Допустим, у нас есть следующая модель:

```
public class Book
    {
        public int Id { get; set; }
        public string Name { get; set; }

    }

public class Author
```

```
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public ICollection<Book> Books { get; set; }
    }
```

Модель автора содердит ссылку на коллекцию книг. В контроллере мы получаем эту модель:

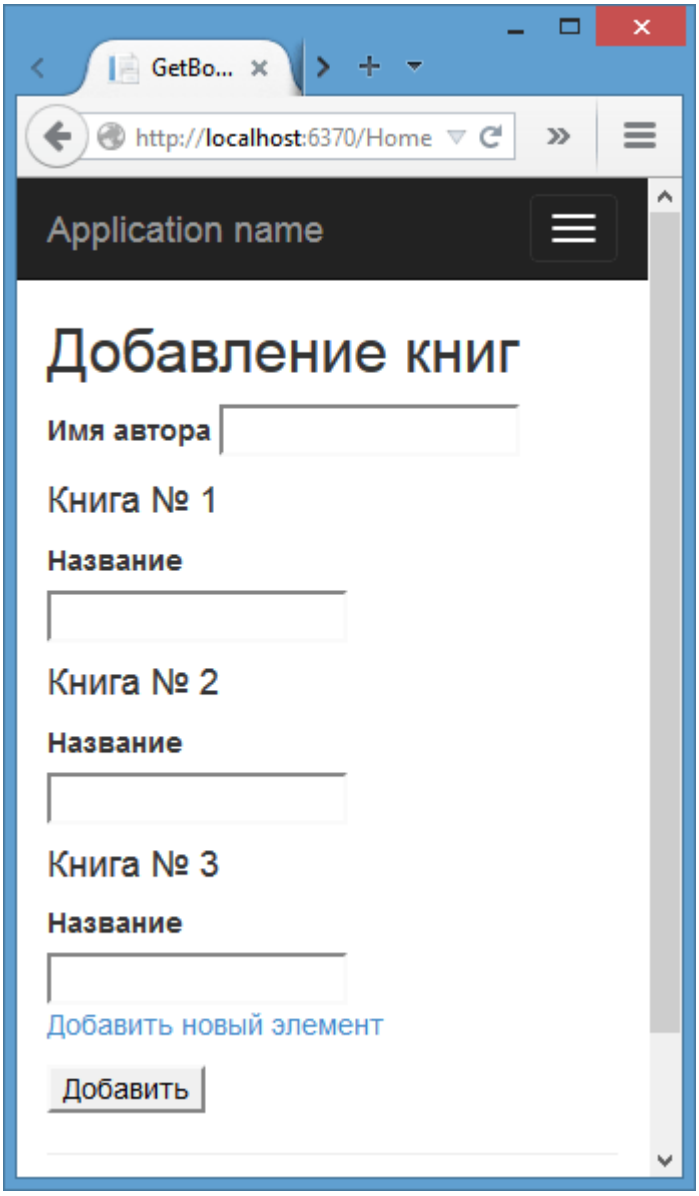
```
[HttpPost]
public ActionResult GetAuthor(Author author)
        {
            return View();
        }
```

Тогда представление могло бы выглядеть так:

```
@model ArrayPostApp.Models.Author

<h2>Добавление книг</h2>

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()
    <div class="authorBlock">
        <label>Имя автора</label>
        <input type="text" name="name" />
    </div>
    <div id="booksBlock">
        <div class="bookItem">
            <h4>Книга № 1</h4>
            <div>
                <label>Название</label>
                <div>
                    <input type="text" name="Books[0].name" />
                </div>
            </div>
        </div>

    </div>
    <p><a class="addLink">Добавить новый элемент</a></p>
    <p><input type="submit" value="Добавить" /></p>
}

@section Scripts {
    <script>
        $(function () {
            var i = 0;
```

```
            $('.addLink').click(function () {
                i++;
                var html2Add = "<div class='bookItem'>" +
             "<h4>Книга № " + (i + 1) + "</h4>" +
             "<div><label>Название</label><div>" +
                    "<input type='text' name='Books[" + i + "].name' />"
+
                 "</div></div></div>";
                $('#booksBlock').append(html2Add);
            })
        })
    </script>
}
```

Так как модель **Author** содержит набор книг в свойстве **Books**, то атрибут **name** соответствующих текстовых элементов должно иметь следующую форму: **name="Books[0].name"**. С помощью кода **javascript** мы можем динамически добавить новые элементы:


GetBo...
http://localhost:6370/Home
Application name
Добавление книг
Имя автора
Книга № 1
Название
Книга № 2
Название
Книга № 3
Название
Добавить новый элемент
Добавить

Миграция баз данных

Нередко возникает ситуация, когда модель меняется. Например, мы решили внести в нее новые свойства. Но при этом у нас уже имеется существующая база данных, в которой есть какие-то данные. И чтобы без потерь обновить базу данных **ASP.NET MVC** предлагает нам такой механизм как миграции. Например, у нас есть простая модель User:

```
public class User
    {
        public int Id { get; set; }
        public string Name { get; set; }
    }
```

Соответсвенно есть контекст данных, через который мы работаем с **БД**:

```
public class UserContext : DbContext
    {
        public UserContext() :
            base("DefaultConnection")
        { }
        public DbSet<User> Users { get; set; }
    }
```

И допустим, у нас есть вся инфраструктура для работы с этой моделью - представления, контроллеры, и у нас есть уже в базе данных несколько объектов данной модели. Но в какой-то момент мы решили изменить модельную базу приложения. Например, мы добавили еще одно поле в модель **User**:

```
public class User
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public int Age { get; set; }
    }
```

Кроме того, мы решили добавить еще одну модель, например:

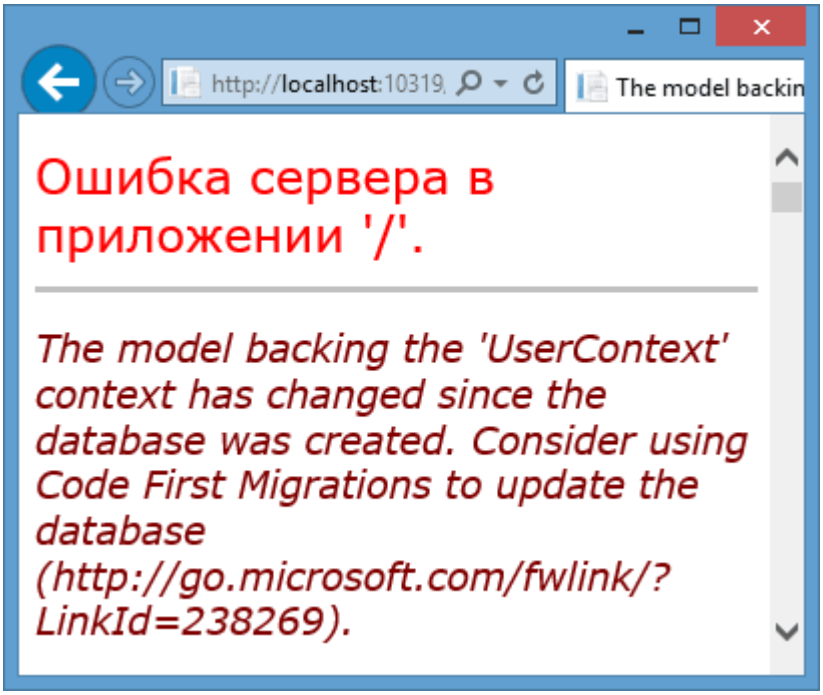
```
public class Company
    {
        public int Id { get; set; }
        public string Name { get; set; }
    }
```

Таким образом, контекст данных у нас уже меняется следующим образом:

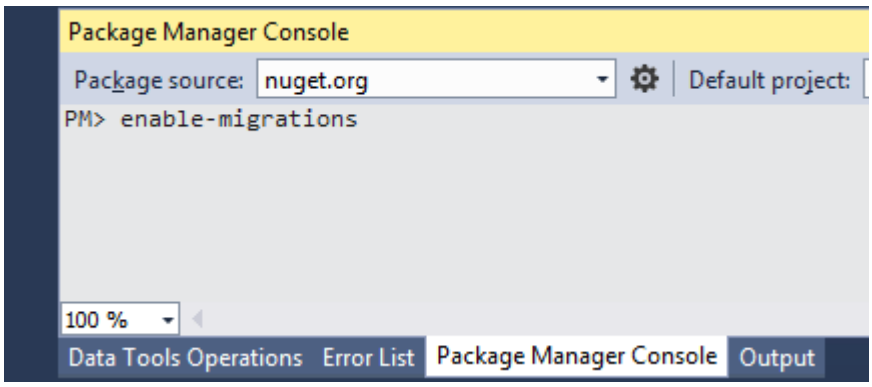
```
public class UserContext : DbContext
```

```
{
    public UserContext() :
        base("DefaultConnection")
    { }
    public DbSet<User> Users { get; set; }
    public DbSet<Company> Companies { get; set; }
}
```

Мы можем добавить в представления для модели **User** дополнительное поле для свойства **Age**, можем создать контроллер и представления для новой модели, но при попытке добавить новый объект в бд, мы будем получать ошибку:



Контекст данных изменился, и теперь нам надо провести миграцию от старой схемы базы данных к новой. И первым делом найдем внизу **Visual Studio** окно **Package Manager Console**, введем в нем команду: **enable-migrations** и нажмем **Enter**:



После выполнения этой команды **Visual Studio** в проекте будет создана папка **Migrations**, в которой можно найти файл **Configuration.cs**. Этот файл содержит объявление одноименного класса **Configuration**, который устанавливает настройки конфигурации:

```
namespace MUser.Migrations
{
    using System;
    using System.Data.Entity;
    using System.Data.Entity.Migrations;
    using System.Linq;

    internal sealed class Configuration :
DbMigrationsConfiguration<MUser.Models.UserContext>
    {
        public Configuration()
        {
            AutomaticMigrationsEnabled = false;
        }

        protected override void Seed(MUser.Models.UserContext context)
        {
            //  This method will be called after migrating to the latest
version.

            //  You can use the DbSet<T>.AddOrUpdate() helper extension
method
            //  to avoid creating duplicate seed data. E.g.
            //
            //    context.People.AddOrUpdate(
            //      p => p.FullName,
            //      new Person { FullName = "Andrew Peters" },
            //      new Person { FullName = "Brice Lambson" },
            //      new Person { FullName = "Rowan Miller" }
            //    );
            //
        }
    }
}
```

В методе Seed можно проинициализировать базу данных начальными данными. Теперь нам надо создать саму миграцию. Там же в консоли **Package Manager Console** введем команду:

PM> Add-Migration "MigrateDB"

После этого **Visual Studio** автоматически сгенерирует класс миграции:

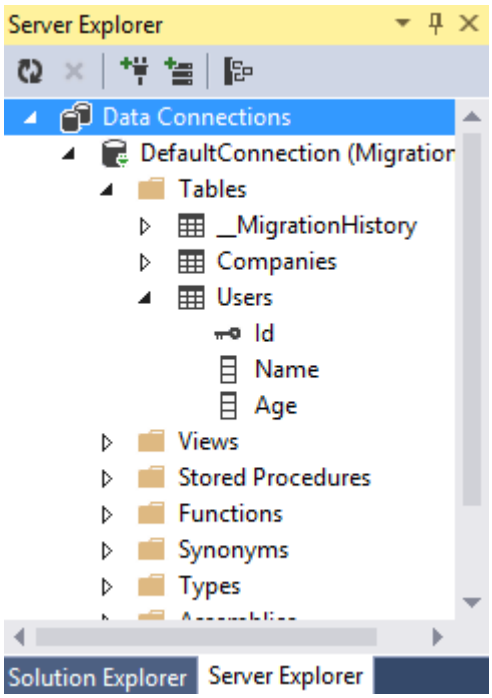
```
namespace MUser.Migrations
{
    using System;
    using System.Data.Entity.Migrations;

    public partial class MigrateDB : DbMigration
    {
        public override void Up()
        {
            CreateTable(
                "dbo.Companies",
                c => new
                    {
                        Id = c.Int(nullable: false, identity: true),
                        Name = c.String(),
                    })
                .PrimaryKey(t => t.Id);

            CreateTable(
                "dbo.Users",
                c => new
                    {
                        Id = c.Int(nullable: false, identity: true),
                        Name = c.String(),
                        Age = c.Int(nullable: false),
                    })
                .PrimaryKey(t => t.Id);

        }

        public override void Down()
        {
            DropTable("dbo.Users");
            DropTable("dbo.Companies");
        }
    }
}
```

В методе **Up** с помощью вызова метода **CreateTable** создается таблица "**dbo.Companies**" и производится ее настройка: создание столбцов, установка ключей. И также добавляется новый столбец **Age** в уже имеющуюся таблицу. Метод **Down** удаляет столбец и таблицу на случай, если они существуют. Фактически эти методы равнозначные выражению **ALTER** в языке **SQL**, которое меняет структуру базы данных и ее таблиц.

И в завершении чтобы выполнить миграцию, применим этот класс, набрав в той же консоли команду:

PM> Update-Database

После этого, если мы посмотрим на состав базы данных, то увидим, что к ней были применены изменения в соответствии с выполненной миграцией:



Итак, миграция выполнена, и мы можем уже использовать обновленные модели и контекст данных.

Создание пагинации

Одной из распространенных задач при работе с данными является проблема пагинации или разделения данных на несколько блоков - страниц, которые выводятсяв представление по отдельности и которые сопровождаются удобной навигацией по страницам.

Для создания пагинации мы можем воспользоваться готовыми плагинами. Но в данном случае мы сами создадим механизм для постраничного вывода.

Вначале опрелелим модель, данные которой будем выводить:

```
public class Phone
    {
        public int Id { get; set; }
        public string Model { get; set; }
        public string Producer { get; set; }
    }
```

Также определим модель, которая будет описывать механизм пагинации, а также дополнительную модель для вывода данных в представление:

```
public class PageInfo
    {
        public int PageNumber { get; set; } // номер текущей страницы
        public int PageSize { get; set; } // кол-во объектов на странице
        public int TotalItems { get; set; } // всего объектов
        public int TotalPages  // всего страниц
        {
            get { return (int)Math.Ceiling((decimal)TotalItems /
PageSize); }
        }
    }

public class IndexViewModel
    {
        public IEnumerable<Phone> Phones { get; set; }
        public PageInfo PageInfo { get; set; }
    }
```

Так как вместе с данными моделей телефонов потребуется также использовать в представлении модель пагинации, то они инкапсулируются классом **IndexViewModel**.

Теперь определим контроллер и его метод для вывода данных:

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

```
    {
        List<Phone> phones;
        public HomeController()
        {
            phones = new List<Phone>();
            phones.Add(new Phone { Id = 1, Model = "Samsung Galaxy III",
Producer = "Samsung" });
            phones.Add(new Phone { Id = 2, Model = "Samsung Ace II",
Producer = "Samsung" });
            phones.Add(new Phone { Id = 3, Model = "HTC Hero", Producer =
"HTC" });
            phones.Add(new Phone { Id = 4, Model = "HTC One S", Producer
= "HTC" });
            phones.Add(new Phone { Id = 5, Model = "HTC One X", Producer
= "HTC" });
            phones.Add(new Phone { Id = 6, Model = "LG Optimus 3D",
Producer = "LG" });
            phones.Add(new Phone { Id = 7, Model = "Nokia N9", Producer =
"Nokia" });
            phones.Add(new Phone { Id = 8, Model = "Samsung Galaxy
Nexus", Producer = "Samsung" });
            phones.Add(new Phone { Id = 9, Model = "Sony Xperia X10",
Producer = "SONY" });
            phones.Add(new Phone { Id = 10, Model = "Samsung Galaxy II",
Producer = "Samsung" });
        }


        public ActionResult Index(int page = 1)
        {
            int pageSize = 3; // количество объектов на страницу
            IEnumerable<Phone> phonesPerPages = phones.Skip((page - 1) *
pageSize).Take(pageSize);
            PageInfo pageInfo = new PageInfo { PageNumber = page,
PageSize = pageSize, TotalItems = phones.Count };
            IndexViewModel ivm = new IndexViewModel { PageInfo =
pageInfo, Phones = phonesPerPages };
            return View(ivm);
        }

    }
}
```

Для простоты примера я создаю обычный список в конструкторе контроллера, однако это мог бы быть и вывод из базы данных.

Метод **Index** в качестве параметра принимает номер страницы. По умолчанию номер страницы будет равняться единице.

В самом методе с помощью комбинации методов **Skip** и **Take** происходит отбор нужной части данных: метод **Skip()** пропускает определенное количество данных, которое передается в параметре, а метод **Take()** извлекает определенное количество.

Для создания механизма пагинации в представлении удобно использовать **хелперы**. Создадим собственный хелпер. Для этого добавим в проект папку **Helpers** и затем добавим в нее следующий класс:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Web;
using System.Web.Mvc;
using ManualPaginApp.Models;

namespace ManualPaginApp.Helpers
{
    public static class PagingHelpers
    {
        public static MvcHtmlString PageLinks(this HtmlHelper html,
        PageInfo pageInfo, Func<int, string> pageUrl)
        {
            StringBuilder result = new StringBuilder();
            for (int i = 1; i <= pageInfo.TotalPages; i++)
            {
                TagBuilder tag = new TagBuilder("a");
                tag.MergeAttribute("href", pageUrl(i));
                tag.InnerHtml = i.ToString();
                // если текущая страница, то выделяем ее,
                // например, добавляя класс
                if (i == pageInfo.PageNumber)
                {
                    tag.AddCssClass("selected");
                    tag.AddCssClass("btn-primary");
                }
                tag.AddCssClass("btn btn-default");
                result.Append(tag.ToString());
            }
            return MvcHtmlString.Create(result.ToString());
        }
    }
}
```

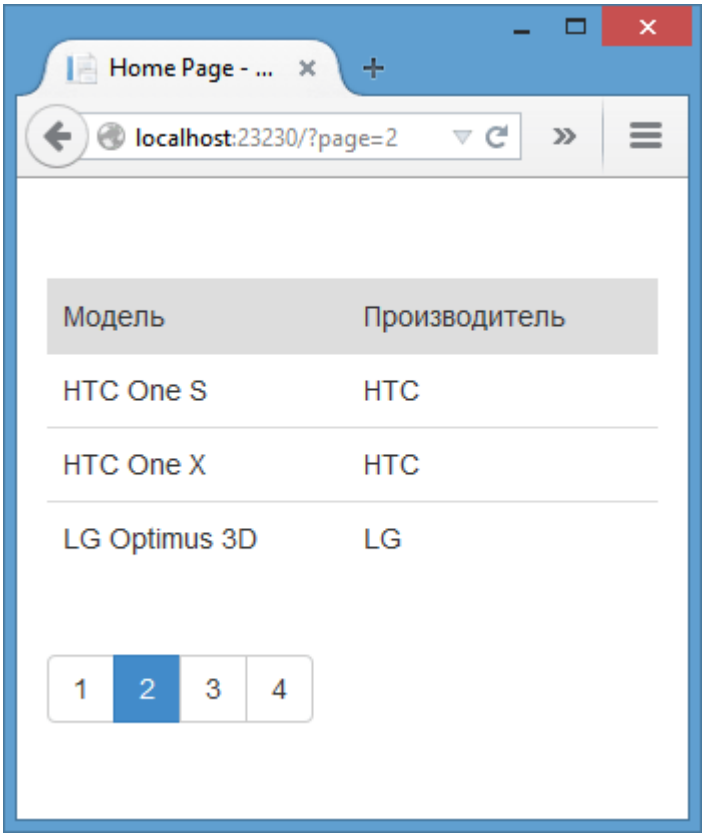
Данный хелпер просто создет блок ссылок, а также добавляет им классы для визуализации. Классы могут быть любыми, но в данном случае использовались стандартные классы **bootstrap**.

И теперь определим представление, которое использовать пагинацию:

```
@model ManualPaginApp.Models.IndexViewModel
@using ManualPaginApp.Helpers
@{
    ViewBag.Title = "Home Page";
}
<table class="table">
    <tr>
        <td><b>Модель</b></td>
        <td><b>Производитель</b></td>
    </tr>
    @foreach (var item in Model.Phones)
    {
        <tr>
            <td>@item.Model</td>
            <td>@item.Producer</td>
        </tr>
    }
</table>
<br />
<div class="btn-group">
    @Html.PageLinks(Model.PageInfo, x => Url.Action("Index", new { page =
x }))
</div>
```

Поскольку хелпер пагинации находится в папке **Helpers**, то необходимо вначале представления подключить данное пространство имен **using ManualPaginApp.Helpers**.

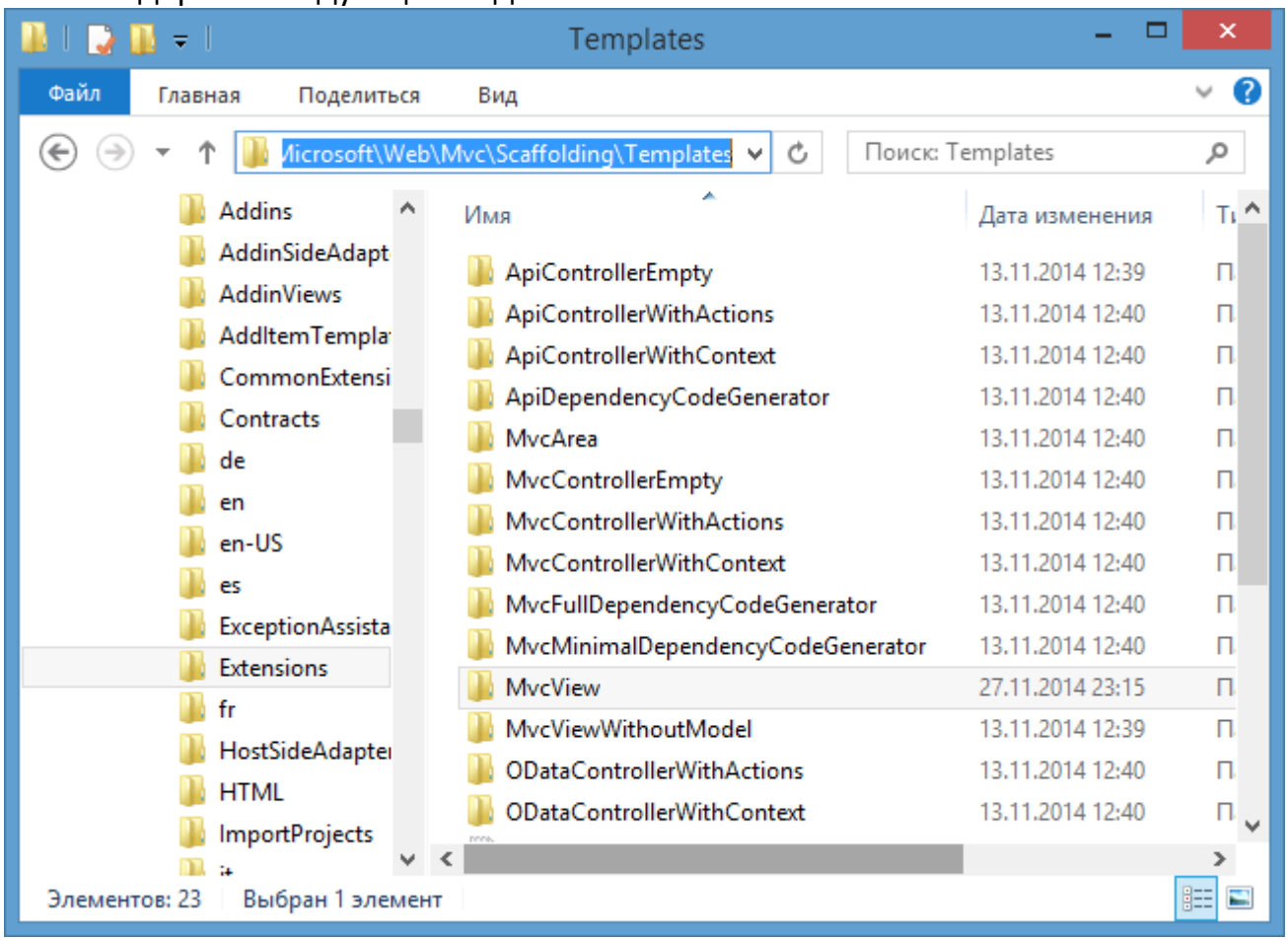
И хелпер пагинации **Html.PageLinks** сделают всю работу по формированию блока ссылок в сответствии с заложенной в нем логикой.
В итоге получим примерно следующую картину:


Home Page - ...
localhost:23230/?page=2
Модель
Производитель
HTC One S
HTC
HTC One X
HTC
LG Optimus 3D
LG
1
2
3
4

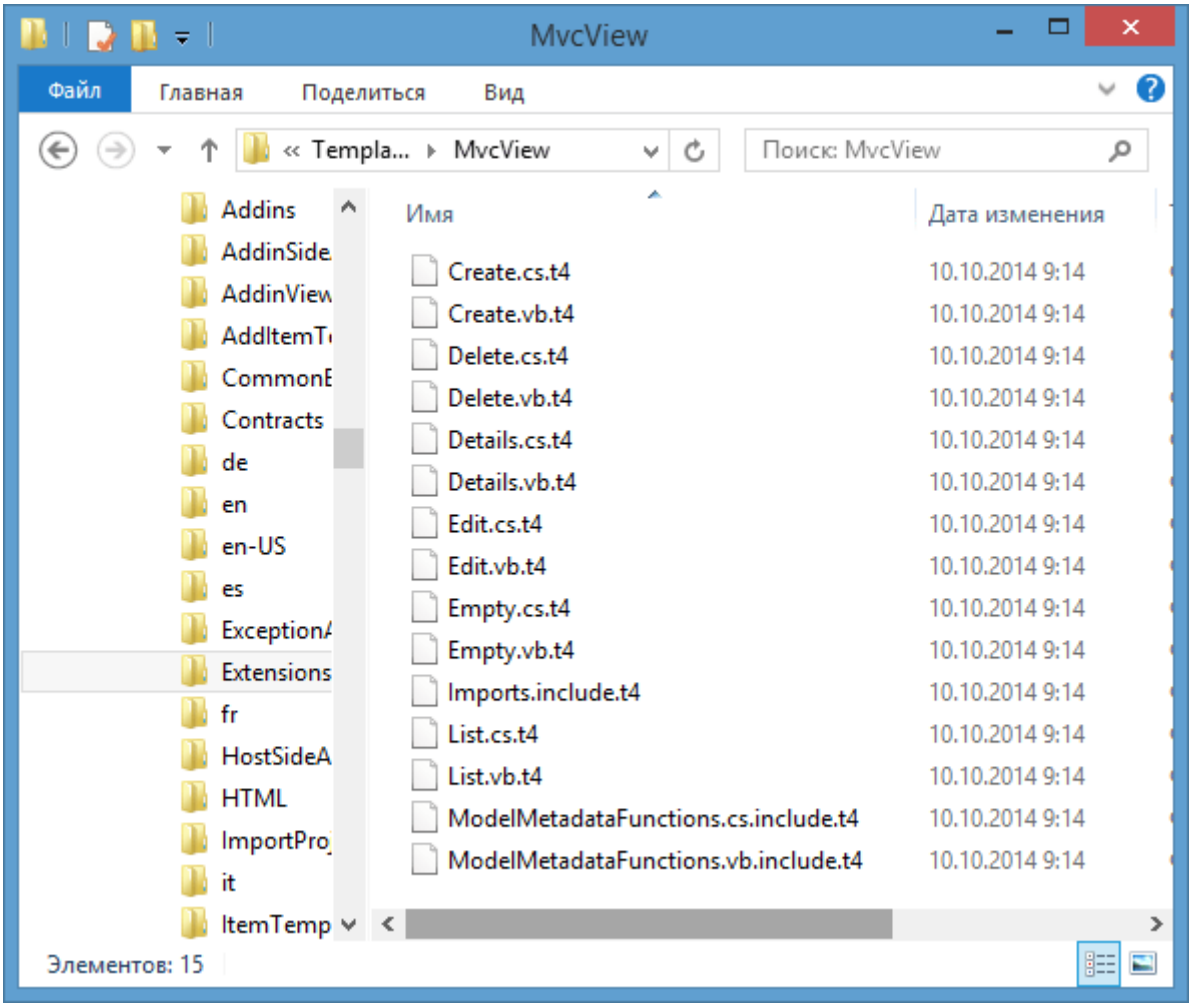
Переопределение шаблонов формирования

Ранее мы рассмотрели функциональность шаблонов формирования, которые позволяют автоматизировать процесс создания контроллера и представлений. Однако иногда их функциональности бывает недостаточно, и возникает необходимость переопределить их поведение.

Для начала посмотрим, что представляют имеющиеся шаблоны. Все шаблоны для **Visual Studio 2013** хранятся в папке ***C:\Program Files (x86)\Microsoft Visual Studio 12.0\Common7\IDE\Extensions\Microsoft\Web\Mvc\Scaffolding\Templates***. Эта папка содержит следующие подкаталоги:



Все подкаталоги представляют шаблоны, на основе которых генерируется код контроллеров или представлений. В частности, каталог **MvcView** содержит шаблоны для генерации представлений:



Шаблон представляет собой файл с расширением **.t4**, который содержит ряд директив, по которым затем генерируется код. Например, наиболее маленький шаблон **Empty.cs.t4** имеет следующий код:

```
<#@ template language="C#" hostspecific="True" #>
    <#@ output extension=".cshtml" #>
        <#@ include file="Imports.include.t4" #>
            @model <#= ViewDataTypeName #><# =ViewDataTypeName #>
                <# />/ The following chained if-statement outputs the
file header code and markup for a partial view, a view using a layout
page, or a regular view.
                if(IsPartialView) {
                #>

                <# } else if(islayoutpageselected) {
                   #>

                    @{
    ViewBag.Title = "<#= ViewName#>";
    <# =ViewName#>
```

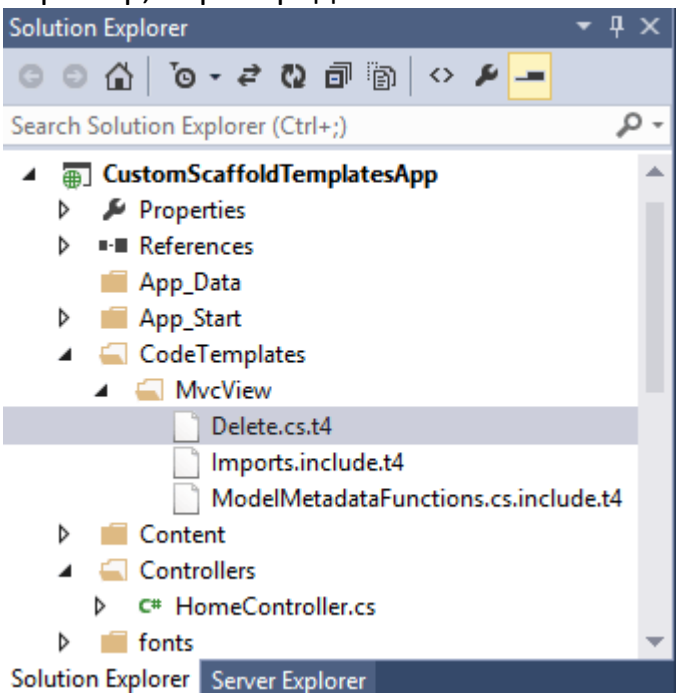
```
        <# if (!string.isnullorempty(layoutpagefile)) {
           #>
            Layout = "<# =LayoutPageFile#>
                ";
                <# }
                   #>
                    }

                    <h2><# =ViewName#></h2>

                    <# } else {
                       #>

                        @{
    Layout = null;
                        }

                        <!DOCTYPE HTML>

                        <html>
                        <head>
                            <meta name="viewport" content="width=device-
width" />
                            <title><# =ViewName #></title>
                        </head>
                        <body>
                            <# pushindent(" ");
                               }
                               #>
                                <# if(!ispartialview &&
!islayoutpageselected) {
                                   #>
                                    <div>
                                    </div>
                                    <# }
                                       #>
                                        <# />/ The following code closes
the tag used in the case of a view using a layout page and the body and
html tags in the case of a regular view page
                                        #>
                                        <# if(!ispartialview &&
!islayoutpageselected) {
                                           clearindent();
                                           #>

</body>
</html>
<# }
#>
```

Все директивы и управляющие блоки шаблонов помещаются между тегами **<#** и **#>**. Например, директива **<#@ output extension=".cshtml" #>** указывает, что выходной файл после обработки должен иметь расширение **.cshtml**.

Блоки обычного текста располагаются вне тегов **<#** и **#>**. Блоки текста не обрабатываются и помещаются в выходной файл как есть.

Мы можем изменить какой-либо шаблон из существующих, однако изменение затронет все последующие случаи использования данного шаблона. И чтобы не менять глобально, мы можем добавить нужные шаблоны в проект в **Visual Studio**. Шаблоны из проекта в **Visual Studio** переопределяют поведение глобальных шаблонов из каталога **C:\Program Files (x86)\Microsoft Visual Studio 12.0\Common7\IDE\Extensions\Microsoft\Web\Mvc\Scaffolding\Templates** для данного проекта.

Чтобы переопределить шаблон, нам надо создать в проекте папку с именем **CodeTemplates**. А в нее затем поместить те шаблоны, которые надо переопределить. Например, переопределим шаблон **Delete.cs.t4**:



Поскольку шаблон **Delete.cs.t4** находится в папке **MvcView**, то в прокте надо создать соответствующую папку в **CodeTemplates** и затем скопировать туда шаблон. Кроме того, я добавил еще пару файлов, от которых зависит **Delete.cs.t4** и которые также находятся в папке **C:\Program Files (x86)\Microsoft Visual Studio 12.0\Common7\IDE\Extensions\Microsoft\Web\Mvc\Scaffolding\Templates\MvcView**

Теперь изменим код шаблона **Delete.cs.t4** на следующий:

```
<#@ template language="C#" hostspecific="True" #>
<#@ output extension=".cshtml" #>
<#@ include file="Imports.include.t4" #>
<h2>Шаблон удаления</h2>
```

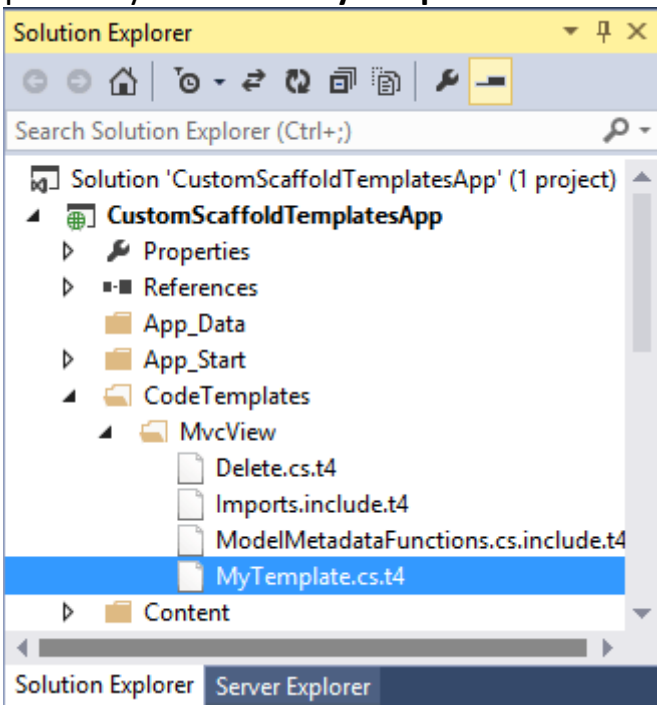
Этот шаблон по сути ничего не делает кроме того, что выводит заголовок.

Первая директива указывает, что будет использоваться язык **C#**. Вторая строка говорит, что выходной файл будет иметь расширение **.cshtml**, а третья строка подключает файл.

Теперь, если мы попытаемся создать представление по шаблону **Delete**, то **Visual Studio** сгенерирует файл со следующим содержанием:

```
<h2>Шаблон удаления</h2>
```

Теперь создадим уже свой более осмысленный шаблон. Для этого добавим в проект в папку **CodeTemplates/MvcView** какой-нибудь файл или другой шаблон и переименуем его на **MyTemplate**:

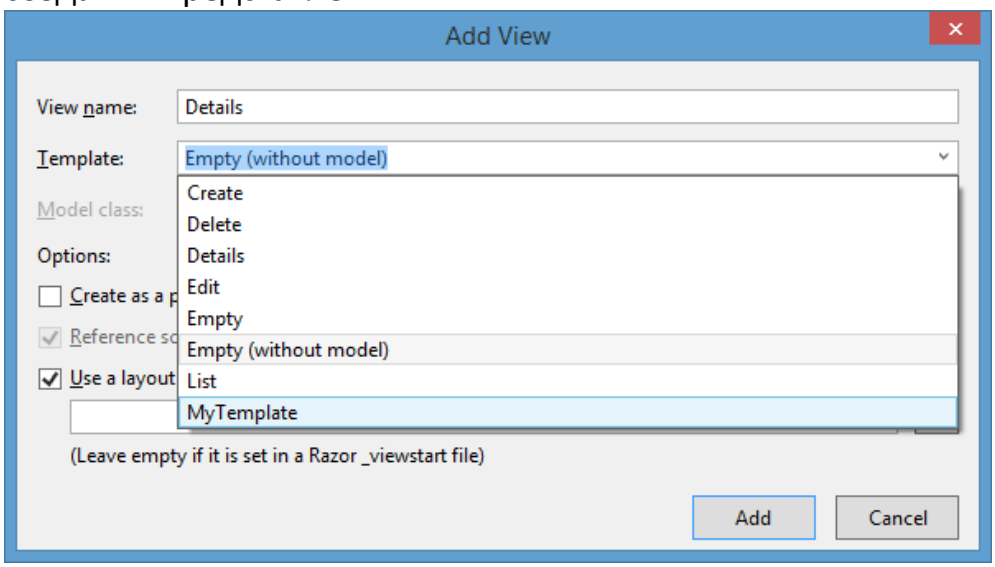


Определим в нем следующий код:

```
<#@ template language="C#" hostspecific="True" #>
    <#@ output extension=".cshtml" #>
        <#@ include file="Imports.include.t4" #>
            @model <#= ViewDataTypeName #><# =ViewDataTypeName #>

                <!DOCTYPE HTML>
                <html>
                <head>
                    <meta name="viewport" content="width=device-width" />
```

```
                    <title>MyTemplate</title>
                </head>
                <body>
                    <h2><# =ViewName#></h2>
                    <table>
                        <# foreach (propertymetadata property in
modelmetadata.properties) {
                            if (property.scaffold &&
!property.isprimarykey && !property.isforeignkey) {
                            #>
                             <tr>
                                 <td>
                                     <b> @Html.DisplayNameFor(model =>
model.<#= GetValueExpression(property) #>)<#
=GetValueExpression(property) #> </b>
                                 </td>

                                 <td>
                                     @Html.DisplayFor(model => model.<#=
GetValueExpression(property) #>)<# =GetValueExpression(property) #>
                                 </td>
                             </tr>
                             <# }
                                }
                               #>
        </table>
    </body>
 </html>
<#@ include file="ModelMetadataFunctions.cs.include.t4" #>
```

Этот шаблон принимает модель из контроллера и выводит все значения ее свойств в виде таблицы. Теперь мы можем использовать этот шаблон при создании представлений:



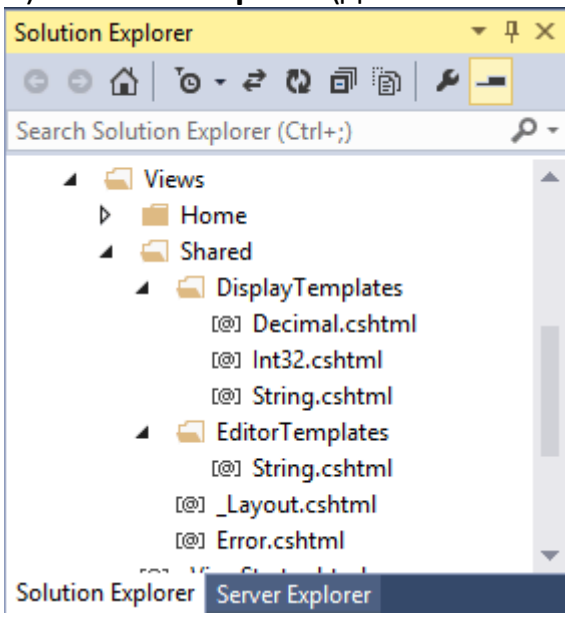
Переопределение шаблонов отображения и редактирования

При использовании методов **DisplayFor()/EditorFor()** фреймворк **MVC** сам определяет, каким образом и какую разметку **html** создавать для отображения и редактирования полей модели. Но мы можем переопределить подобное поведение, указав, как именно **MVC** должен работать с отдельными типами данных.

Допустим, у нас есть следующая модель:

```
public class Book
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public decimal Price { get; set; }
    }
```

Модель содержит свойства трех типов данных: **int, string, decimal**. Переопределим создание разметки для этих свойств. Для этого добавим в проект в каталог **Views/Shared** две новых папки: **DisplayTemplates** (для шаблонов отображения) и **EditorTemplates**(для шаблонов редактирования):



Так как класс использует свойства трех разных типов данных, в шаблонах отображения в папке **DisplayTemplates** я создал по файлу для каждого типа. Названия файлов носят названия классов, которые представляют данный тип. Например, файл **Int32.cshtml**, который используется для визуализации значений **int**, имеет следующую разметку:

```
@model Int32

<b>@Model</b>
```

Директива `@model Int32` указывает, что в качестве модели будет использоваться тип **int** или **Int32**. А само содержимое модели делается жирным шрифтом с помощью тега **<b>**
Файл **String.cshtml** имеет следующее содержимое:

```
@model String

<b>"@Model"</b>
```

Тут тоже самое, только название указывается в кавычках. В общем-то мы можем определить разные теги или структуру тегов, можно, например, задать классы (**"@Model"**) и т.д.
Файл **Decimal.cshtml**:

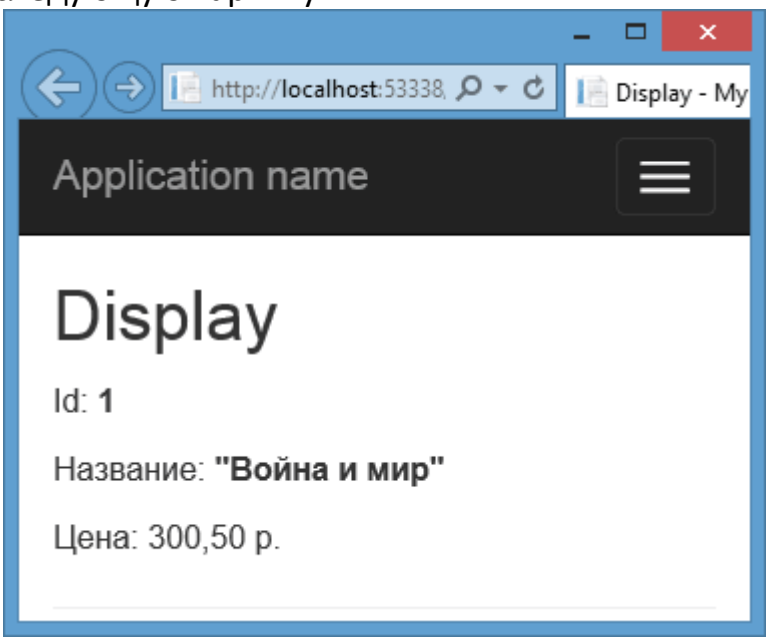
```
@model decimal
@{
    IFormatProvider formatProvider =
        new System.Globalization.CultureInfo("ru-RU");
        <span class="currency">@Model.ToString("C",
formatProvider)</span>
}
```

Здесь уже определяется логика форматирования значения в качестве денежной единицы. Теперь если мы используем хелпер **DisplayFor()** для вывода свойств модели:

```
@model CustomScaffoldTemplatesApp.Models.Book

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>MyTemplate</title>
</head>
<body>
<h2>Display</h2>
<table>
     <tr>
        <td>
           <b> @Html.DisplayNameFor(model => model.Name) </b>
        </td>

        <td>
            @Html.DisplayFor(model => model.Name)
        </td>
     </tr>
```

```
        <tr>
            <td>
                <b> @Html.DisplayNameFor(model => model.Price) </b>
            </td>

            <td>
                @Html.DisplayFor(model => model.Price)
            </td>
        </tr>
</table>
</body>
</html>
```

(**http://localhost:40674/Home/Display/2**)
То мы получим следующую картину:



В папку **EditorTemplates** я добавил только один шаблон форматирования - **String.cshtml**:

```
@model string

<input type="text" name="name" style="border-color:red; height:40px;
background-color:#eee;" value="@Model" />
```

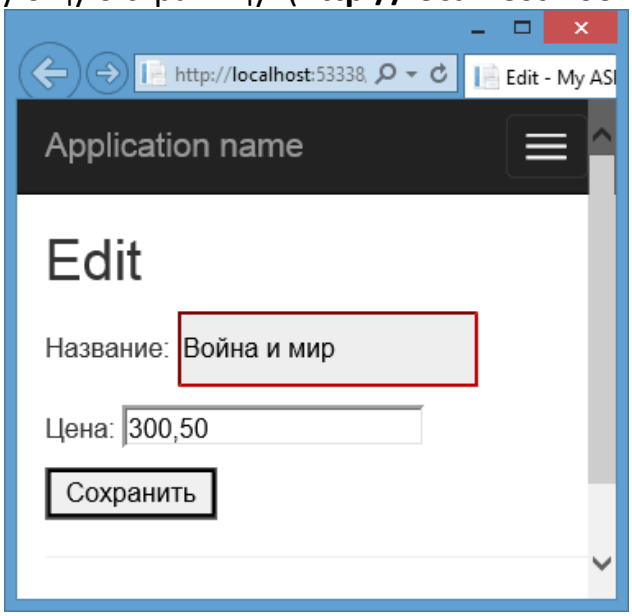
И допустим, у нас есть следующее представление для редактирования данных:

```
public ActionResult Edit(int id)
        {
            if (id == null)
            {
                return HttpNotFound();
```

```
            }
            Book book = db.Books.Find(id);
            if (book != null)
            {
                return View(book);
            }
            return HttpNotFound();
        }
```

```
@model CustomScaffoldTemplatesApp.Models.Book

@{
    ViewBag.Title = "Edit";
}

<h2>Edit</h2>

@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()
    @Html.HiddenFor(model => model.Id)
    <p>Название: @Html.EditorFor(model => model.Name)</p>
    <p>Цена: @Html.EditorFor(model => model.Price)</p>
    <p><input type="submit" value="Сохранить" /></p>
}
```

То мы получим следующую страницу: (**http://localhost:40674/Home/Edit/2**)



При создании подобных шаблонов на основе элементов ввода следует учитывать их ограниченность: в данном случае элемент имеет атрибут **name="name"**, поэтому его значение будет сопоставляться со свойством **Name**. Если у нас она модель в которой есть только одно строковое свойство, которое называется **Name**, то подобный подход еще сработает. Но у нас может быть несколько свойств с типом **string**, которые могут называться по-разному. В этом случае мы можем определить шаблон вывода или редактирования модели целиком.

Например, добавим в папку **DisplayTemplates** следующий файл **Book.cshtml**:

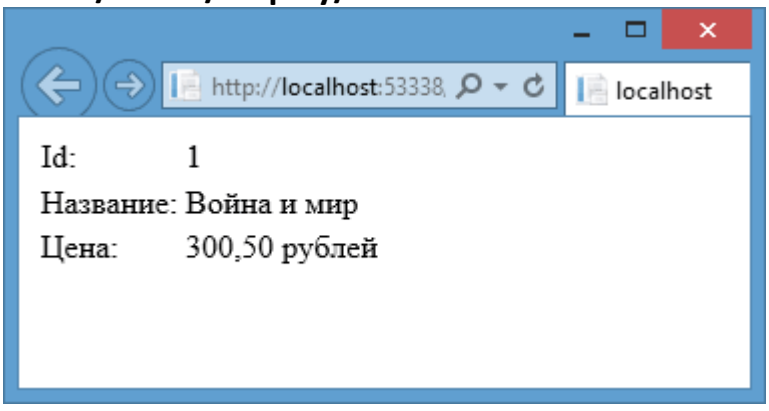
```
@model CustomScaffoldTemplatesApp.Models.Book

<table>
    <tr><td>Id:</td><td>@Model.Id</td></tr>
    <tr><td>Название:</td><td>@Model.Name</td></tr>
    <tr><td>Цена:</td><td>@Model.Price рублей</td></tr>
</table>
```

CustomDisplayEditorTemplates - в данном случае это название моего проекта, в котором определен класс **Book**.

Значения всех свойств модели выводятся через таблицу. Тогда представление для отображения модели будет выглядеть так:

```
@model CustomScaffoldTemplatesApp.Models.Book
@{
    Layout = null;
}
@Html.DisplayForModel()
```

http://localhost:40674/Home/Display/2



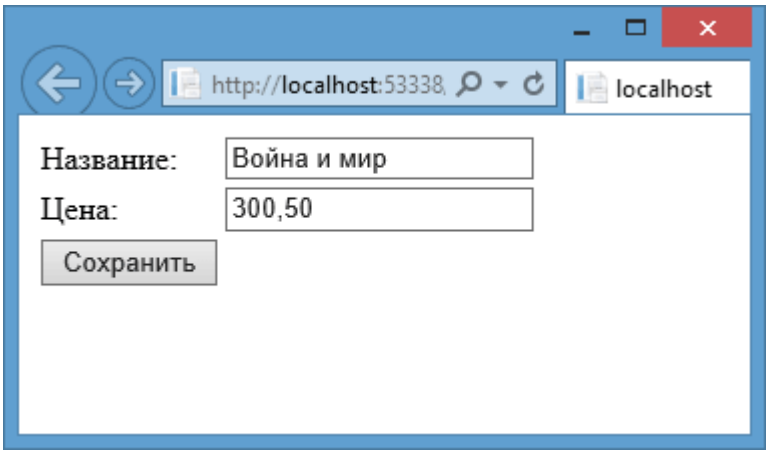
Также добавим в каталог **EditorTemplates** шаблон для редактирования **Book.cshtml**:

```
@model CustomScaffoldTemplatesApp.Models.Book

@using (Html.BeginForm())
```

```
{
    @Html.HiddenFor(model => model.Id)
    <table>
        <tr><td>Название: </td><td><input type="text" name="name"
value="@Model.Name" /></td></tr>
        <tr><td>Цена: </td><td><input type="text" name="price"
value="@Model.Price" /></td></tr>
        <tr><td><input type="submit" value="Сохранить"
/></td><td></td></tr>
    </table>
}
```

Используем этот шаблон для создания формы редактирования:

```
@model CustomScaffoldTemplatesApp.Models.Book

@{
    Layout = null;
}
@Html.EditorForModel()
```



Маршрутизация

Определение маршрутов

В предыдущих главах мы так или иначе сталкивались с маршрутизацией. Например, при обращении к некоторому действию контроллера мы набирали в адресной строке браузера **http://localhost:3456/Home/Index**, где **Home** - имя контроллера без префикса **Controller**, а **Index** - имя метода действия этого контроллера. Если метод **Index** принимал бы какой-нибудь параметр, например, типа **int**: **public ActionResult Index(int Id)**, то мы могли обратиться к этому методу и передать значение в его параметр с помощью следующей строки: **http://localhost:3456/Home/Index/5**. В этом плане все очень просто. Но мы не говорили еще о том, почему мы должны прописывать маршрут именно так, и как мы собственно можем управлять маршрутами.

Рассмотрим механизм определения маршрутов. В **ASP.NET MVC 5** все определения маршрутов находятся в файле **RouteConfig.cs**, который располагается в проекте в папке **App_Start**. Если мы на него посмотрим, то увидим настройки маршрута по умолчанию:

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

namespace CustomScaffoldTemplatesApp
{
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new { controller = "Home", action = "Index", id
= UrlParameter.Optional }
            );
        }
    }
}
```

Установкой маршрутов занимается статический метод **RegisterRoutes**. Однако на этом определение маршрутов не заканчивается, так как нам еще

надо вызвать этот метод в приложении при запуске. Это производится в файле **Global.asax** в методе **Application_Start**:

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

namespace CustomScaffoldTemplatesApp
{
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);
        }
    }
}
```

Вызов метода `RouteConfig.RegisterRoutes(RouteTable.Routes);` как раз и производит регистрацию маршрутов в приложении.

Теперь разберем определение маршрута по умолчанию в файле **RouteConfig.cs**. Первая строка

```
routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
```

отключает обработку запросов для некоторых файлов, например с расширением ***.axd (WebResource.axd)**.

Далее идет собственно определение маршрута по умолчанию. Метод **routes.MapRoute** выполняет сопоставление маршрута запросу. В перегруженных версиях данного метода мы можем задать дополнительные параметры сопоставления.

Разберем параметры метода. Вначале с помощью свойства **name** задается имя маршрута - **Default**. Второй параметр - **url** задает шаблон строки запроса или **шаблон Url**, с которым будет сопоставляться данный маршрут.

Шаблон **URL** включает в себя несколько сегментов, которые заключены в фигурные скобки. В данном случае сегмент представляет часть запроса, которая находится между слешами. Каждый такой сегмент шаблона содержит параметр. Эти параметры называются параметрами **URL**. В данном случае это параметры **controller, action** и **id**. Но вообще параметры могут иметь различные имена, включающие любые алфавитно-цифровые символы.

При получении запроса механизм маршрутизации парсит строку **URL** и помещает значения маршрута в словарь - в **объектRouteValueDictionary**, доступный через контекст приложения **RequestContext**. В качестве ключей в нем применяются имена параметров **URL**, а соответствующие сегменты **URL** выступают в качестве значений. Например, у нас есть следующий **URL** запроса: **http://localhost/Home/Index/5**, то в этом случае образуются следующие пары ключей и значений в словаре **RouteValueDictionary**:

Параметр	Значение
controller	Home
action	Index
id	5

Третий параметр метода **routes.MapRoute - defaults** определяет значения по умолчанию для маршрута. Например, если вдруг в строке запрос указаны не все параметры, а сам запрос, к примеру, идет по адресу **http://localhost/**, то система маршрутизации вызовет метод **Index** контроллера **Home**, как указано в параметре **defaults**. Также, если мы не укажем метод контроллера, например, **http://localhost/Home/,** также будет вызван метод иконтроллера **Home**.

Поэтому если мы захотим, к примеру, чтобы у нас по умолчанию клиент обращался не к методу **Index** контроллера **HomeController**, а, например, к методу **Show** контроллера **BookController**, то мы можем соответственно изменить значения данного параметра:

```
defaults: new { controller = "Book", action = "Show", id = UrlParameter.Optional }
```

Последний параметр объявлен как необязательный `id = UrlParameter.Optional`, поэтому, если он не указан в строке запроса, он не будет учитываться и передаваться в словарь параметров **RouteValueDictionary**.

Например, запрос **http://localhost/Home/Create/3** вызовет метод **Create** контроллера **HomeController**, передав в этот метод в качестве параметра число **3**. В то же время запрос **http://localhost/Home/Create/** также вызовет метод **Create** контроллера **HomeController**, хотя последний параметр в нем не указан.

Таким образом, настройки по умолчанию позволяют нам не указывать в строке запроса полностью название контроллера и его метода. Но в случае, если такие настройки не заданы, мы должны определять в строке запроса контроллер и его метод. Например, изменим установку маршрутов в файле **RouteConfig.cs** следующим образом:

```
public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
```

```
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}");
        }
    }
```

Теперь обратимся к нашему приложению, набрав в строке браузера только адрес сайта, например, **http://mysyte.com/**. В этом случае мы получим информацию об ошибке. Ошибка будет состоять в том, что теперь нам надо полностью набирать в строке запроса адрес ресурса. Поэтому следующий адрес **http://mysyte.com/Home/Index** будет нормально работать (если, конечно, в приложении определен контроллер **Home** с методом **Index** и соответствующим ему представлением).

И если мы теперь перейдем по адресу **http://localhost/Home/**, как мы это делали выше, то опять получим ошибку, так как в строке запроса указан только один сегмент. А в определении маршрута у нас указано два сегмента - **{controller}/{action}**. Если для параметров не определены значения по умолчанию, то строка запроса должна иметь такое же число сегментов, для которых не определены значения по умолчанию.

В то же время если запрос будет состоять из трех сегментов, например, **http://localhost/Home/Index/1**, то мы также получим ошибку, потому что число сегментов в запросе больше числа, определенного в шаблоне **URL** данного маршрута.

Работа с маршрутами

Создание новых маршрутов

Для создания маршрутов можно использовать метод **MapRoute**. Например:

```
public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
            name: "newRoute",
            url: "{controller}/{action}");
        }
    }
```

Либо можно сначала создать объект **Route**, а потом добавить его в коллекцию маршрутов **RouteCollection**. Определим два маршрута:

```
public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}/{id}");

            Route newRoute = new Route("{controller}/{action}", new
MvcRouteHandler());
            routes.Add(newRoute);

        }
    }
```

Итак, здесь определены два маршрута. Первый - маршрут **Default** сопоставляется с запросами, имеющими три сегмента. Второй – **newRoute** сопоставляется с запросами, имеющими только два сегмента. Так, вызов **http://localhost:5555/Home/Index/1** будет соответствовать первому маршруту, так как в нем определено три сегмента.

А вызов **http://localhost:5555/Home/Index** - второму маршруту. Вызов **http://localhost:5555/Home** не будет соответствовать ни одному маршруту, так как у нас не определен маршрут, принимающий только один сегмент в шаблоне **URL**.

Правда, ситуация с этими двумя запросами в большей степени является искусственной, так как мы можем совместить их, просто определив параметр **id** как необязательный:

```
routes.MapRoute(name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new { id = UrlParameter.Optional });
```

При передаче значений по умолчанию для параметров важно учитывать позицию параметра. Движок маршрутизации использует значение по умолчанию только в том случае, если все последующие параметры также имеют значения по умолчанию. Так, если мы зададим следующий маршрут:

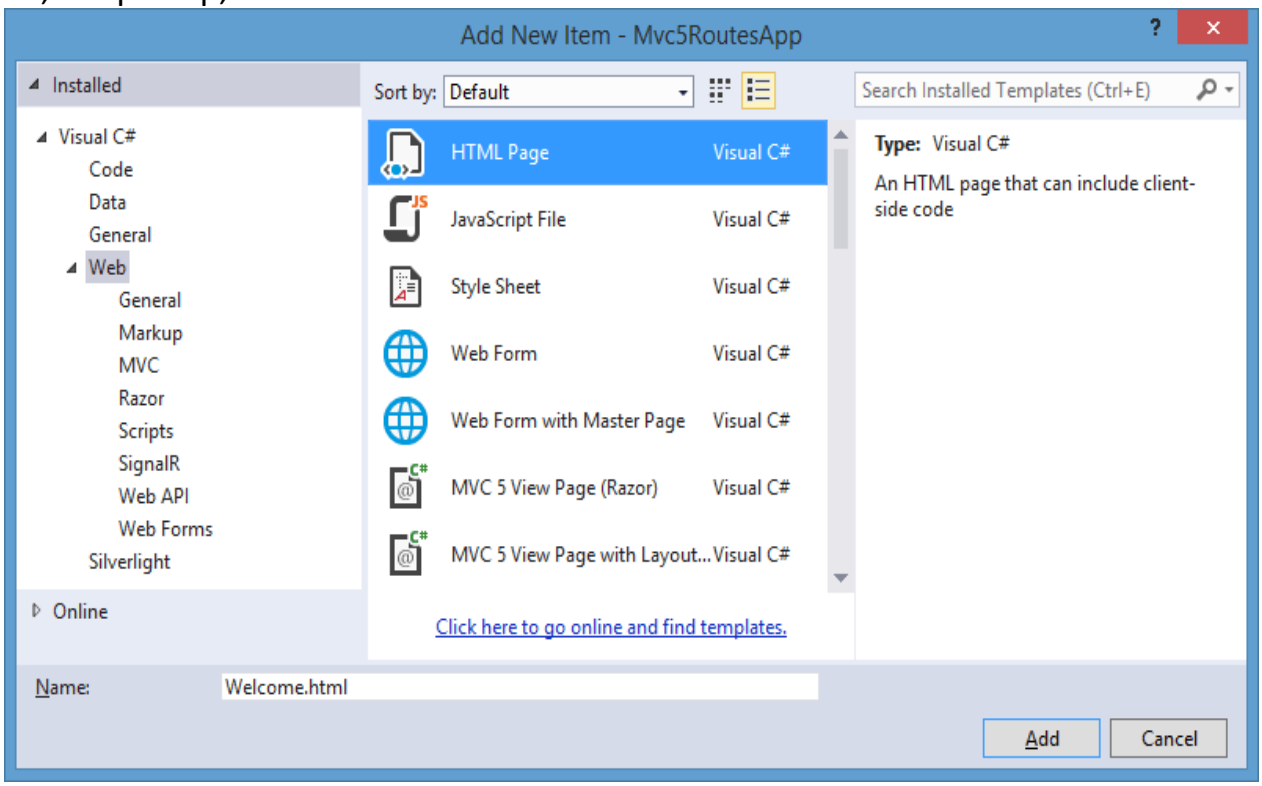
```
routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new { action = "Index" });
```

то запрос, содержащий два сегмента, например, **Home/2** не будет сопоставляться с данным маршрутом. Поэтому нам надо в данном случае указать значение по умолчанию также и для параметра **id**.

Сопоставление запросов с файлами на диске

Мы посмотрели, как сопоставлять маршруты с контроллерами и их методами. Однако мы можем адресовать запросы также и отдельным файлам сайта, например, статическим **html**-страницам. Работа механизма маршрутизации такова, что сначала он смотрит, совпадает ли запрос с определенным файлом, хранящимся на сервере, и если такого файла не находит, тогда он начинает сопоставлять запрос с определенными маршрутами.

Итак, добавим в папку **Content** проекта новую **html-страницу** и назовем ее, например, **Welcome.html**



Для тестирования создадим какой-нибудь контент на этой странице. И после ее добавления в проект мы можем к ней обращаться напрямую через запрос **Content/Welcome.html**.

Использование префиксов в строке запроса

Сегменты строки запроса необязательно должны нести только значения для параметров, определенных в маршруте. Можно также использовать различные префиксы в строке запроса и соответствующим образом настроить

маршрут для обработки подобных запросов. Например, мы хотим, чтобы запрос содержал префикс **Uz: http://localhost:49326/Ru/Home/Index/1**.

Тогда обрабатывающий этот запрос маршрут может выглядеть следующим образом:

```
routes.MapRoute(
                name: "Default",
                url: "Ru/{controller}/{action}/{id}",
                defaults: new { id = UrlParameter.Optional }
            );
```

Кроме того, можно добавлять префиксы не в качестве отдельного сегмента, а к самому сегменту:

```
routes.MapRoute(
                name: "Default",
                url: "Ru{controller}/{action}/{id}",
                defaults: new { id = UrlParameter.Optional });
```

В этом случае строка запроса, соответствующая данному маршруту, может выглядеть так: **http://localhost:49326/RuHome/Index**

Порядок определения новых маршрутов

При добавлении новых маршрутов важно учитывать их порядок. Обычно более специфические маршруты помещаются перед более общими. Например, предположим, мы определили следующие маршруты:

```
public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new { id = UrlParameter.Optional });

            routes.MapRoute(
                name: "Default2",
                url: "Ru{controller}/{action}/{id}",
                defaults: new { id = UrlParameter.Optional });
        }
```

При отправлении приложению запроса типа **http://localhost:49326/RuHome/Index**, если у нас в приложении не определен контроллер **RuHomeController**, приложение вернет ошибку.

Почему? Потому система маршрутизации пытается сопоставить запрос сначала с первым маршрутом. Если запрос не соответствует первому маршруту

- тогда со вторым и так далее по списку маршрутов, пока не найдет нужный. В данном случае входящий запрос соответствует и первому, и второму маршрутам. Однако первый маршрут будет искать контроллер по имени **RuHomeController** и, не найдя его, вернет ошибку. Поэтому чтобы подобная ситуация не произошла, надо сначала определить маршрут **Default2**, который более специфичен и не конфликтует с первым:

```
public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                name: "Default2",
                url: "Ru{controller}/{action}/{id}",
                defaults: new { id = UrlParameter.Optional });

            routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new { id = UrlParameter.Optional });
        }
```

Также мы можем использовать префиксы в качестве псевдонимов для контроллера и его метода:

```
routes.MapRoute(
                name: "Default2",
                url: "Store/Buy",
                defaults: new { controller = "Home", action = "Index" });
```

Здесь для контроллера **HomeController** используется псевдоним **Store**, а для действия **Index** - псевдоним **Buy**. В итоге данный маршрут будет сопоставляться с таким запросом, как **Store/Buy**. А система маршрутизации будет обращаться по такому запросу к методу **Index** контроллера **Home**.

Получение переданных параметров

Чтобы получить переданные значения для параметров маршрута, мы можем воспользоваться объектом **RouteData**. Например, если у нас определен стандартный маршрут

```
routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new { controller = "Home", action = "Index", id
= UrlParameter.Optional }
            );
```

то мы можем получить значение параметра **controller** следующим образом (**http://localhost:34362/Home/GetParam**):

```
public string GetParam()
        {
            string controller =
RouteData.Values["controller"].ToString();
            return controller;
        }
```

Значения, передаваемые в качестве дополнительных значений, например, для параметра **id**, можно извлекать из самого параметра метода:

```
public ActionResult GetParam(int id)
        {
            ViewBag.OldId = id;
            //или так
            // ViewBag.OldId = RouteData.Values["id"];
            return View();
        }
```

Передача произвольного количества параметров в запросе

Ранее мы ограничивались только тремя сегментами. Но если у нас скажем метод принимает два и более параметров:

```
public ActionResult MyMethod(int id = 1, string name = "")
        {
            ViewBag.Name = name;
            return View();
        }
```

То мы просто можем добавить в маршрут нужное нам количество параметров:

```
routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}/{name}",
                defaults: new { id = UrlParameter.Optional, name =
UrlParameter.Optional });
```

Тогда данный маршрут будет обрабатывать запрос типа **Home/Index/1/name**, что эквивалентно следующей строке запроса: **Home/Index?id=1&name=name**

Кроме того, мы можем обозначить любое количество сегментов в запросе, чтобы не быть жестко привязанным к числу сегментов с помощью параметра **{*catchall}**:

```
routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}/{*catchall}",
```

```
                defaults: new { controller = "Home", action = "Index", id
= UrlParameter.Optional });
```

Теперь мы можем обрабатывать запросы с любым количеством сегментов:

Запрос	Параметры запроса
mysyte.com	controller=Home action=Index
mysyte.com/Book	controller=Book action=Index
mysyte.com/Book/Show	controller=Book action=Show
mysyte.com/Book/Show/2	controller=Book action=Show id=2
mysyte.com/Book/Show/2/Oldedition	controller=Book action=Show id=2 catchall=Oldedition
mysyte.com/Book/Show/2/Oldedition/1960	controller=Book action=Show id=2 catchall=Oldedition/1960

После получения значения для параметра **catchall** мы сами должны обработать его и получить уже значения для отдельных сегментов.

Создание ограничений для маршрутов

Иногда возникает необходимость, более точно задать совпадение строки запроса для данного маршрута. Например, нам надо задать, чтобы имя контроллера обязательно начиналось с буквы "**H**". Тогда нам надо определить соответствующее ограничение для имени с помощью регулярных выражений:

```
routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new { controller = "Home", action = "Index", id
= UrlParameter.Optional },
                constraints: new { controller = "^H.*" }
            );
```

С помощью параметра **constraints** устанавливаются ограничения маршрута. И если мы направим приложению запрос **Book/Index**, даже если у

нас есть контроллер **BookController** с методом **Index**, то приложение вернет ошибку, так как имя контроллера попадает под ограничение, а другого маршрута, которому бы соответствовал запрос **Book/Index**, у нас не задано.

Подобным образом мы можем задать ограничения и для других параметров. Например, пусть параметр **Id** состоит как минимум из двух цифр:

```
routes.MapRoute(
              name: "Default",
              url: "{controller}/{action}/{id}/{*catchall}",
              defaults: new { controller = "Home", action = "Index" },
              constraints: new { controller = "^H.*", id = @"\d{2}" }
                );
```

Теперь даже запрос **Home/Index/1** у нас не будет сопоставляться с маршрутом, так как параметр **Id** состоит из одной цифры, а не из двух.

Также мы можем задать ограничение по типу метода. Например, мы хотим, чтобы обрабатывались только запросы для методов с типом GET:

```
routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}/{*catchall}",
                defaults: new { controller = "Home", action = "Index" },
                constraints: new { controller = "^H.*", id =
@"\d{2}",httpMethod = new HttpMethodConstraint("GET") }
                );
```

Теперь будут обрабатываться только те запросы, которые представляют запросы типа **GET**.

Создание собственных ограничений

Чтобы создать собственное ограничение, нужно реализовать интерфейс **IRouteConstraint** с одним единственным методом **Match**, который имеет следующее определение:

```
public interface IRouteConstraint
        {
            bool Match(HttpContextBase httpContext, Route route, string
parameterName,
                RouteValueDictionary values, RouteDirection
routeDirection);
        }
```

Ограничение маршрута применяет этот интерфейс **IRouteConstraint**. Это вынуждает движок маршрутизации вызвать для ограничения маршрута метод **IRouteConstraint.Match**, чтобы определить, применяется ли данное ограничение к данному запросу или нет. Например, создадим ограничение,

которое не будет пропускать запросу по некоторому **url**. Итак, добавим в приложение следующий класс:

```
public class CustomConstraint : IRouteConstraint
        {
            private string uri;
            public CustomConstraint(string uri)
            {
                this.uri = uri;
            }
            public bool Match(HttpContextBase httpContext, Route route,
string parameterName,
                    RouteValueDictionary values, RouteDirection
routeDirection)
            {
                return !(uri == httpContext.Request.Url.AbsolutePath);
            }
        }
```

Здесь мы говорим, что если запрашиваемый ресурс совпадает со значением свойства **httpContext.Request.Url.AbsolutePath**, то запрос не будет сопоставляться с маршрутом. Тогда определение маршрута может выглядеть следующим образом:

```
routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}/{*catchall}",
                defaults: new { controller = "Home", action = "Index" },
                constraints: new { id = @"\d{2}", myConstraint = new
CustomConstraint("/Home/Index/12") }
                );
```

Теперь запрос **/Home/Index/12** не будет обрабатываться, даже если он удовлетворяет всем остальным условиям и ограничениям.

Игнорирование запросов

По умолчанию в методе **RegisterRoutes** класса **RouteConfig** определена такая строка **routes.IgnoreRoute("{resource}.axd/{*pathInfo}");**. С помощью данного выражения мы запрещаем доступ к определенному ресурсу, находящемуся на сервере. Так, мы можем переписать предыдущий пример, где использовали ограничение для маршрута в виде запрета пути **/Home/Index/12**, следующим образом:

```
public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
            routes.IgnoreRoute("Home/Index/12");
            routes.MapRoute(
```

```
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new { controller = "Home", action = "Index", id
= UrlParameter.Optional }
            );
        }
```

Теперь также запрос **Home/Index/12** не будет обрабатываться и сопоставляться с определенным ресурсом на сервере.

Генерация исходящих адресов URL

Еще одной стороной механизма маршрутизации является генерации исходящих адресов в приложении, например, при выводе их в представлении. Наиболее простой способ вывода адреса является использование анкора - элемента **a**:

```
<a href="Home/Index/3" />
```

Но также мы можем использовать специальные хелперы рендеринга - **Html.ActionLink** и **Html.RouteLink**

Html.ActionLink

Хелпер **ActionLink** создает гиперссылку на действие контроллера. Если мы создаем ссылку на действие, определенное в том же контроллере, то можем просто указать имя действия:

```
@Html.ActionLink("Жми здесь", "Show")
```

Что создаст вам следующую разметку:

```
<a href="/Home/Show">Жми здесь</a>
```

Когда надо указать ссылку на действие из другого контроллера, то в хелпере **ActionLink** в качестве третьего аргумента имя контроллера. Например, ссылка на действие **List** контроллера **Book** будет создаваться так:

```
@Html.ActionLink("Список книг", "List", "Book")
```

Кроме того, если у нас в некотором методе **Index** контроллера **Book** определено несколько параметров:

```
public class BookController : Controller
        {
            public string Index(string author = "Толстой", int id = 1)
            {
                return author + "  " + id.ToString();
            }
        }
```

То перегруженная версия хелпера **ActionLink** позволяет передать параметр объекта (обычно анонимный тип) для параметра **routeValues**. Среда выполнения принимает свойства объекта и использует их для создания значений маршрутизации (имена свойств становятся именами параметров маршрута, а значения свойств представляют значения параметра маршрута). Создадим ссылку для вышеопределенного действия контроллера:

```
@Html.ActionLink("Все книги", "Index", "Book", new { id = 10 }, null)
//или
@Html.ActionLink("Достоевский", "Index", "Book", new { author =
"Достоевский", id = 5 }, null)
```

Последний параметр в данном хелпере является параметром **htmlAttributes**. Мы можем использовать этот параметр для установки значения атрибута элемента **HTML**. В данном случае передается значение **null** (то есть никаких атрибутов не устанавливается).

Теперь попробуем передать атрибуты, например, установить атрибуты **id** и **class**:

```
@Html.ActionLink("Все книги", "Index", "Book", new { author = "Толстой",
id = 10 }, new { id = "Tolstoi", @class = "link" })
```

Сгенерированная **html**-разметка будет выглядеть следующим образом:

```
<a class="link"
href="/Book/Index/10?author=%D0%A2%D0%BE%D0%BB%D1%81%D1%82%D0%BE%D0%B9"
id="Tolstoi">Все книги</a>
```

Обратите внимание на знак **@** перед словом **class**: поскольку слово "**class**" является зарезервированным словом в **C#**, то для правильного рендеринга нам надо перед ним указать знак **@**.

Html.RouteLink

Хелпер **RouteLink** использует похожий шаблон, что и **ActionLink**: он принимает имя маршрута, но не требует аргументов для имени контроллера и имени действия. Так, первый пример с **ActionLink** эквивалентен следующему коду:

```
@Html.RouteLink("Все книги", new { controller = "Book", action = "Index",
author = "Толстой", id = 10 }, new { id = "Tolstoi", @class = "link" })
```

Чтобы использовать маршрут, нам просто надо указать имя определенного нами маршрута и затем определить при необходимости дополнительные параметры. Например, возьмем стандартный маршрут **Default**:

```
routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new { controller = "Home", action = "Index",
id = RouteParameter.Optional });
```

Тогда создать ссылку мы можем, например, так:

```
@Html.RouteLink("Все книги", "Default", new { action = "Show" })
```

URL-хелперы

URL-хелперы похожи на хелперы **ActionLink** и **RouteLink** за тем исключением, что они не возвращают **HTML**, а создают пути **URL** и возвращают их в виде строк. Имеется три типа **URL**-хелперов:

- **Action**
- **Content**
- **RouteUrl**

Хелпер **Action** похож на **ActionLink** за тем исключением, что но не возвращает тег якоря. Например, следующий код отображает адрес **URL**, но не саму ссылку:

```
@Url.Action("Index", "Book", new { author = "Толстой", id = 10 }, null)
```

Хелпер **RouteUrl** использует тот же шаблон, что и **Action**, но как и **RouteLink**, принимает имя маршрута и аргументы для параметров маршрута:

```
@Url.RouteUrl(new { controller = "Book", action = "Index", author =
"Толстой", id = 10 })
```

Хелпер **Content** преобразует относительные пути в абсолютные. Пример использования хелпера **Content** можно увидеть в представлении **_Layout**:

```
<script src="@Url.Content("~/Scripts/jquery-1.5.1.min.js")"
        type="text/javascript"></script>
```

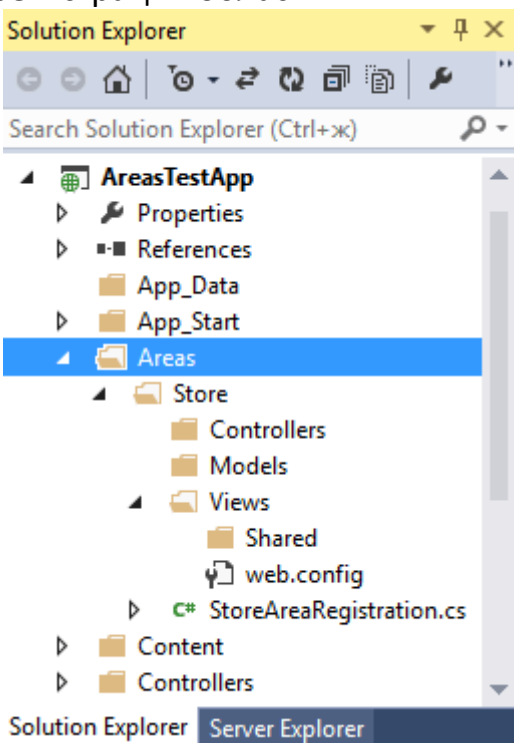
С помощью тильды **(~)** хелпер **Content** генерирует необходимый **URL** относительно расположения приложения. Без тильды **URL** мог бы стать некорректным, если бы вы перенесли приложение в другой виртуальный каталог.

Области

Несмотря на то, что проект **MVC** по умолчанию представляет собой четкую структуру разделения на отдельные функциональные части - контроллеры, модели, представления, иногда для более удобной работы над

приложением, особенно над большими приложениями, приложение делится на ряд областей (area).

Добавим в проект **MVC** область. Нажмем правой кнопкой мыши на проект и в появившемся меню выберем **Add->Area**. В окне добавления области дадим новой области название, например, **Store**. После этого в структуре проекта произойдет ряд изменений: в проект будет добавлена новая папка **Areas**, в которую в свою очередь будет добавлена папка **Store** - непосредственно для нашей новой области. Внутри папки **Store** фактически окажется мини-проект, в котором будут папки для контроллеров, моделей и представлений и класс регистрации области.



Откроем файл регистрации области **StoreAreaRegistration.cs**:

```
using System.Web.Mvc;

namespace AreasTestApp.Areas.Store
{
    public class StoreAreaRegistration : AreaRegistration
    {
        public override string AreaName
        {
            get
            {
                return "Store";
            }
        }
```

```
        public override void RegisterArea(AreaRegistrationContext
context)
        {
            context.MapRoute(
                "Store_default",
                "Store/{controller}/{action}/{id}",
                new { action = "Index", id = UrlParameter.Optional }
            );
        }
    }
}
```

Автоматически сгенерированное определение маршрута в этом файле показывает, как входящие запросы будут сопоставляться с контроллерами и действиями, определенными в данной области **Store**. Однако чтобы сопоставление запросов с областью происходило, также надо зарегистрировать все области в файле **Global.asax**. Правда, вручную не придется это делать, так как при добавлении в проект первой области **Visual Studio** уже автоматически это делает. Вы можете открыть файл **Global.asax** и увидеть изменения:

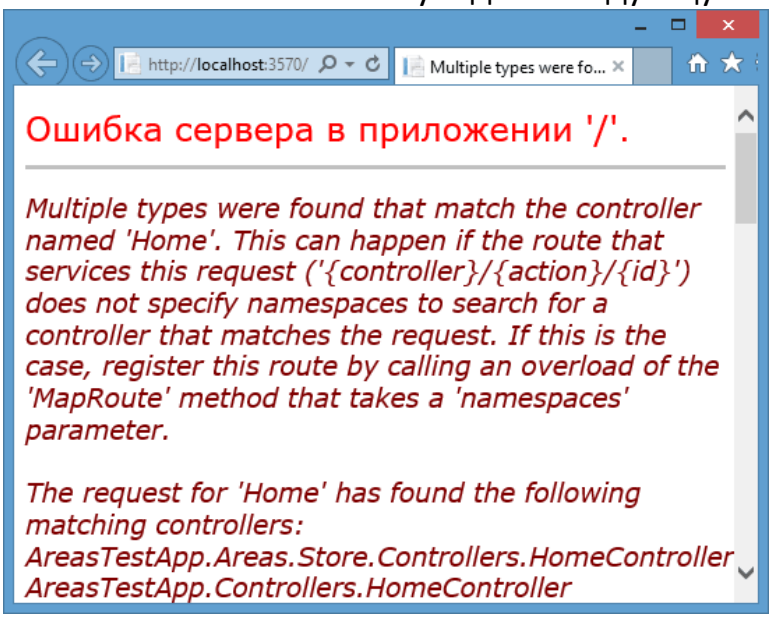
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Optimization;
using System.Web.Routing;

namespace AreasTestApp
{
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);
        }
    }
}
```

Чтобы протестировать нашу область, мы можем также добавить в папку **Controllers**, как и в основном проекте, новый контроллер и определить в нем действие. И также определить для действий контроллера представления. Допустим, у нас в области **Store** определен следующий контроллер **ShopController**:

```
public class ShopController : Controller
    {
        //
        // GET: /Store/Shop/
        public ActionResult Index()
        {
            return View("ShopController");
        }
     }
```

Тогда при запуске приложения мы можем обратиться к методу **Index** контроллера по адресу **/Store/Shop/Index**, указав сначала имя области, а потом как обычно имя контроллера и его метода.

Но что будет, если мы захотим добавить в область контроллер **Home** с методом **Index**, как и в основном приложении. Так как у нас определен стандартный маршрут, который при запуске приложения будет отсылать нас к методу **Index** контроллера **Home**, то система маршрутизации окажется в двойственном положении: она не будет знать, к какому именно контроллеру обращаться - к тому, который определен в основном приложении, или к тому, который определен в области. В итоге вы увидите следующую картину:



Чтобы избежать подобной двойственности, в файле **RouteConfig.cs** проекта надо указать пространство имен контроллера **Home**, который будет вызываться при запуске приложения:

```
public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
```

```
            routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new { controller = "Home", action = "Index", id
= UrlParameter.Optional },
                namespaces: new[] { "AreasTestApp.Controllers" }
            );
        }
    }
```

В данном случае основное приложение по умолчанию находится в пространстве имен **AreasTestApp** (так как проект называется **AreasTestApp**), а контроллер **Home** - в пространстве имен **AreasTestApp.Controllers**. Если бы мы хотели, чтобы у нас при запуске приложения отрабатывал метод **Index**, определенный в контроллере **Home** в области **Store**, то мы могли бы указать соответствующее пространство имен, которое в моем случае - **AreasTestApp.Areas.Store.Controllers**

Генерация ссылок в областях

При генерации ссылок в представлениях отдельных областей используются все те хелперы, например, **Html.ActionLink**, однако тут есть и некоторые особенности.

Чтобы сгенерировать ссылку на какое-либо действие контроллера, которые находятся внутри области, то мы указываем действие и контроллер (если действие находится в другом контроллере):

```
@Html.ActionLink("Все книги", "Index", "Book", new { id = 10 }, null)
```

В итоге будет сгенерирована ссылка:

```
<a href="Store/Book/Index/10>Все книги</a>
```

Если же требуемое действие и контроллер находятся в другой области, то мы указываем область в параметре хелпера:

```
@Html.ActionLink("Все книги", "List", new { area = "Library", controller
= "Book" })
```

Сгенерированная ссылка будет выглядеть так:

```
<a href="Library/Book/List />Все книги</a>
```

Если же метод и контроллер находятся в основном приложении, то для параметра **area** определяем пустую строку:

```
@Html.ActionLink("Все книги", "Index", new { area = "", controller =
"Home" })
```

Создание собственного обработчика маршрутов

Пред тем как приступить к созданию собственного обработчика маршрутов, посмотрим, что в целом представляет собой процесс маршрутизации.

Процесс маршрутизации состоит из следующих этапов:

1. Модуль **UrlRoutingModule** пытается сопоставить текущий запрос с маршрутами в таблице **RouteTable**.
2. Если сопоставление завершилось удачно, то модуль маршрутизации выбирает обработчик маршрутов сопоставленного маршрута - объект **IRouteHandler**.
3. Затем у объекта **IRouteHandler** вызывается метод **GetHandler**, который возвращает объект **IHttpHandler**, используемый для обработки запроса.
4. У обработчика **IHttpHandler** вызывается метод **ProcessRequest** для обработки запросов.

По умолчанию обработчик запросов или объект IRouteHandler представляет экземпляр класса MvcRouteHandler, который возвращает объект MvcHandler, применяющий интерфейс IHttpHandler. Этот объект MvcHandler отвечает за инициализацию контроллера, который потом уже вызывает одно из своих действий.

Однако мы можем переопределить это поведение, указав при определении маршрута свой обработчик маршрутов. Итак, сначала создадим сам обработчик. Для обработчика создадим какую-нибудь специальную папку в проекте и добавим в нее новый класс. Назовем егоMyRouteHandler:

```
1  using System.Web;
2  using System.Web.Routing;
3
4  namespace Routing.RouteHandlers
5  {
6      public class MyRouteHandler : IRouteHandler
7      {
8          public IHttpHandler GetHttpHandler(RequestContext requ
9          {
10             return new MyHttpHandler();
11         }
12     }
13
14     public class MyHttpHandler : IHttpHandler
15     {
           public bool IsReusable
```

```
16          {
17              get { return false; }
18          }
19
20          public void ProcessRequest(HttpContext context)
21          {
22              context.Response.Write("Инопланетное послание : П
23          }
24      }
25  }
26
```

Как было уже сказано про этапы маршрутизации, нам нужно собственно два класса: класс, реализующий интерфейс IHttpHandler, который и будет обрабатывать запрос; и класс, реализующий интерфейс IRouteHandler, который сопоставляется с маршрутом, и вызывает первый класс.

Теперь в классе RouteConfig пропишем маршрут, который будет обрабатываться нашим обработчиком, не забывая при этом импортировать пространство имен, в котором объявлен наш обработчик:

```
1   public class RouteConfig
2   {
3       public static void RegisterRoutes(RouteCollection routes)
4       {
5           routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
6
7           Route newRoute = new Route("{controller}/{action}", ne
8           routes.Add(newRoute);
9       }
10  }
```

Теперь мы можем адресовать нашему приложению соответствующий запрос, например, *Home/Index* и браузер выведет нам нашу строку с посланием.

Атрибуты маршрутизации

Одним из нововведений MVC 5 стала такая функциональность как атрибуты маршрутизации (attribute routing). Данные атрибуты позволяют сопоставить определенный маршрут с методом контроллера. Например, у нас есть такой метод в контроллере HomeController:

```
public string Test(int id, string name)
{
    return id.ToString() + ". " + name;
}
```

Ему мог бы соответствовать следующий маршрут, определенный в файле RouteConfig.cs:

```
public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

        routes.MapRoute(
            name: "Default2",
            url: "{id}/{name}",
            defaults: new { controller = "Home", action = "Tes
            constraints: new { id="\\d+"}
        );

        routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}/{id}",
            defaults: new { controller = "Home", actio
UrlParameter.Optional }
        );
    }
}
```

Параметр constraints: new { id="\\d+"} ограничивает первый сегмент строки запроса числами, таким образом, у нас не будет конфликта между двумя маршрутами. То есть запрос вида *http://localhost:6392/2/volga* будет сопоставляться с первым маршрутом, а если вместо числа 2 будет идти строка - то со вторым.

Но атрибуты маршрутизации позволяют не определять дополнительный маршрут, а указать сопоставление прямо в коде контроллера:

```
[Route("{id:int}/{name}")]
```

```
public string Test(int id, string name)
{
    return id.ToString() + ". " + name;
}
```

Кроме того, нам надо изменить код в файле RouteConfig.cs следующим образом:

```
public class RouteConfig
{
    public static void RegisterRoutes(RouteCollection routes)
    {
        routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

        routes.MapMvcAttributeRoutes();

        routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}/{id}",
            defaults: new { controller = "Home", actio
UrlParameter.Optional }
        );
    }
}
```

Здесь вместо определения маршрута мы используем метод routes.MapMvcAttributeRoutes();, который подключает в систему маршрутизации приложения функциональность атрибутов маршрутизации.

Мы также можем явно указать имя контроллера или метода или параметра в атрибуте: [Route("Home/{id:int}/{name}")]. Данный маршрут будет соответствовать запросу *Home/4/somename*

Ограничения маршрутов

В определении сегмента id мы использовали ограничение, чтобы явно указать, что этот сегмент должен представлять целое число: id:int. Кроме int мы можем задать еще ряд ограничений по типу:

- **alpha**: соответствует только алфавитным символам латинского алфавита. Например, {id:alpha}
- **bool**: соответствует логическлму значению. Например, {id:bool}
- **datetime**: соответствует значению DateTime. Например, {id:datetime}
- **decimal**: соответствует значению decimal. Например, {id:decimal}
- **double**: соответствует значению double. Например, {id:double}
- **float**: соответствует значению float. Например, {id:float}

- **length**: соответствует строке определенной длины, либо ее длина должна быть в определенном диапазоне. Например,{id:length(5)} или {id:length(5, 15)}
- **long**: соответствует значению long. Например, {id:long}
- **max**: соответствует значению int, которое не больше значения max. Например, {id:max(99)}. Аналогичным образом действует ограничение min, только оно указывает на минимально допустимое значение сегмента.
- **maxlength**: соответствует строке, длина которой не больше определенного значения. Например, {id:maxlength(20)}. Аналогичным образом работает ограничение minlength, указывая на минимально допустимую длину строки
- **range**: указывает на диапазон, в пределах которого должно находиться значение сегмента. Например, {id:range(5, 20)}
- **regex**: соответствует регулярному выражению. Например, {id:regex(^\d{3}-\d{3}-\d{4}$)}

Значения по умолчанию

Как и при определении маршрута, мы можем задать значения для параметров по умолчанию:

```
[Route("{id:int}/{name=volga}")]
public string Test(int id, string name)
{
    return id.ToString() + ". " + name;
}
```

Так, если строка запроса не будет содержать последний параметр, то вместо него будет использоваться строка "volga".

Использование префиксов

Выше приводился пример атрибута маршрутизации с название контроллера в начале: [Route("Home/{id:int}/{name}")]. Но если у нас вдруг есть несколько подобных действий, обращение к которым должно начинаться с "Home", то удобно использовать префиксы:

```
[RoutePrefix("home")]
public class HomeController : Controller
{
    [Route("{id:int}/{name}")]
    public string Test(int id, string name)
    {
        return id.ToString() + ". " + name;
    }
    [Route("{id:int}")]
    public string Sead(int id)
```

```
10      {
11          return id.ToString();
12      }
13      [Route("~/lol/twit/{id:int}")]
14      public string Twit(int id)
15      {
16          return id.ToString();
17      }
18  }
19
```

Теперь запрос к обоим методам должен начинаться с Home: "Home/5/fds" или "Home/5". При этом префикс не обязательно должен совпадать с именем контроллера, а может иметь любое значение.

Последний маршрут устраняет действие префикса с помощью знака тильды (~) в начале маршрута. И чтобы к этому методу обратиться, надо будет использовать запрос *http://localhost:6392/lol/twit/2*.

Маршрутизация и вложенные ресурсы

Нередко в различных соцсетях или других сервисах можно встретить ссылки такого типа *https://twitter.com/SomeName/following* или, например, *https://plus.google.com/u/0/12344566556/posts*. Эти дву ссылки объединяет похожая схема маршрутизации - сначала идет общий путь, затем идентификатор пользователя (SomeName или 12344566556), и далее какое-то свойство, относящееся к пользователю - following (подписки в твиттере) или posts (все посты в Google+). Если мы посмотрим с точки зрения организации данных, то получится, что в некоторый класс пользователя хранит объекты постов или подписок. Таким образом, эти объекты окузываются как бы вложенными в объект пользователя. Посмотрим, как мы сможем создать систему маршрутов, чтобы обращаться к этим объектам как к вложенным и в asp.net mvc.

Допустим, у нас есть следующие классы, которые связаны по внешнему ключу и объекты которых храняться в базе данных:

```
1   public class Book
2   {
3       public int Id { get; set; }
4       public string Name { get; set; }
5
6       public int AuthorId { get; set; }
7       public Author Author { get; set; }
8   }
```

```
9     public class Author
10    {
11        public int Id { get; set; }
12        public string Name { get; set; }
13    }
```

Каждый объект Book хранит ссылку на связанного автора из таблицы авторов. Контекст данных мог бы выглядеть так:

```
1    public class BookContext : DbContext
2    {
3        public DbSet<Book> Books { get; set; }
4        public DbSet<Author> Authors { get; set; }
5    }
```

Теперь наша задача состоит в том, чтобы передать пользователю информацию о какой-либо книге и о ее авторе. Для этого создадим в контроллере HomeController два действия:

```
1     public ActionResult GetBook(int? id)
2     {
3         if (id == null)
4             return HttpNotFound();
5         Book book = db.Books.Include(b => b.Author).FirstOrDefault(b=>b.Id == id);
6         if (book == null)
7             return HttpNotFound();
8         return View(book);
9     }
10    public ActionResult GetAuthor(int? id)
11    {
12        if (id == null)
13            return HttpNotFound();
14        Book book = db.Books.Include(b => b.Author).FirstOrDefault(b=>b.Id == id);
15        if (book == null)
16            return HttpNotFound();
17        return View(book.Author);
18    }
```

Каждое действие получает нужную книгу по id и передает в строго типизированное представление объект книги или объект автора книги.

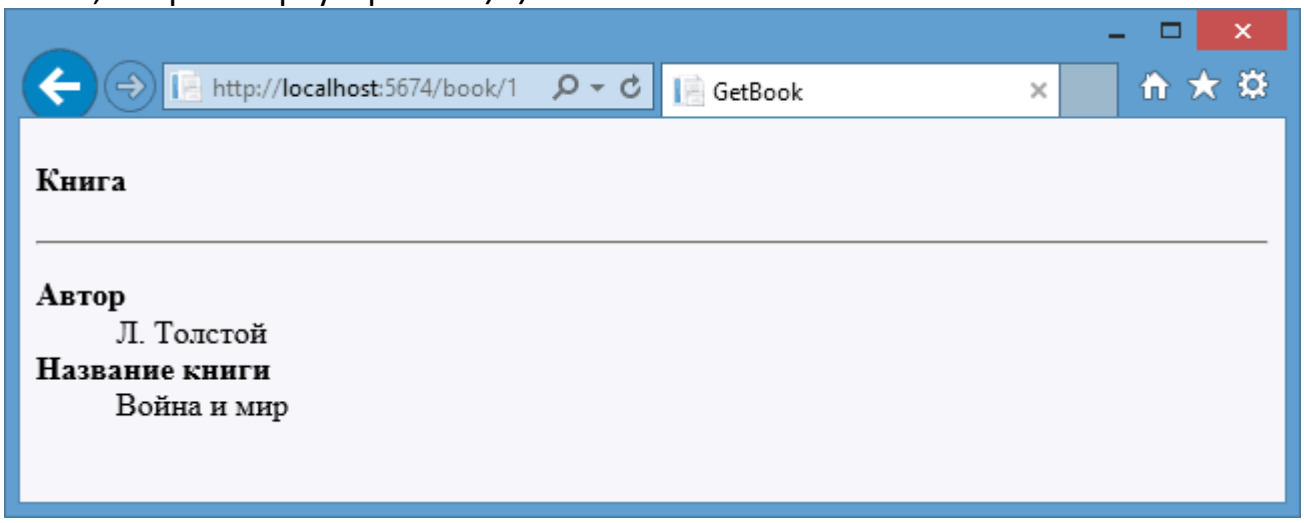
Теперь добавим нужные маршруты. Изменим код в файле RouteConfig.cs следующим образом:

```
1     public class RouteConfig
2     {
3         public static void RegisterRoutes(RouteCollection routes)
```

```
4       {
5           routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
6
7           routes.MapRoute(
8               name: "author",
9               url: "book/{id}/author",
10              defaults: new { controller = "Home", action = "GetAuthor"}
11          );
12
13          routes.MapRoute(
14              name: "book",
15              url: "book/{id}",
16              defaults: new { controller = "Home", action = "GetBook"}
17          );
18
19          routes.MapRoute(
20              name: "Default",
21              url: "{controller}/{action}/{id}",
22              defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
23          );
24      }
25  }
```

Итак, здесь мы добавляем два маршрута. Первый маршрут будет обрабатывать запрос типа *book/1/author* и сопоставлять его с методомGetAuthor (фактически будет представлять запрос *home/getauthor/1*). А второй маршрут будет обрабатывать запрос типа *home/book/1* и сопоставлять его с методом GetBook.

И теперь мы можем запустить приложения и получить информацию о первой книге, набрав в браузере *book/1/*:



А если мы наберем *book/1/author*, то получим информацию об авторе этой книги:

И мы также могли использовать атрибуты маршрутизации, чтобы создать подобные маршруты:

```
1    [Route("book/{id}")]
2    public ActionResult GetBook(int? id)
3    {
4        // остальной код
5    }
6    [Route("book/{id}/author")]
7    public ActionResult GetAuthor(int? id)
8    {
9        // остальной код
10   }
```

А код в файле RouteConfig.cs выглядел бы следующим образом:

```
1
2    public class RouteConfig
3    {
4        public static void RegisterRoutes(RouteCollection routes)
5        {
6            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
7
8            routes.MapMvcAttributeRoutes();
9
10           routes.MapRoute(
11               name: "Default",
12               url: "{controller}/{action}/{id}",
13               defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
14           );
15       }
```

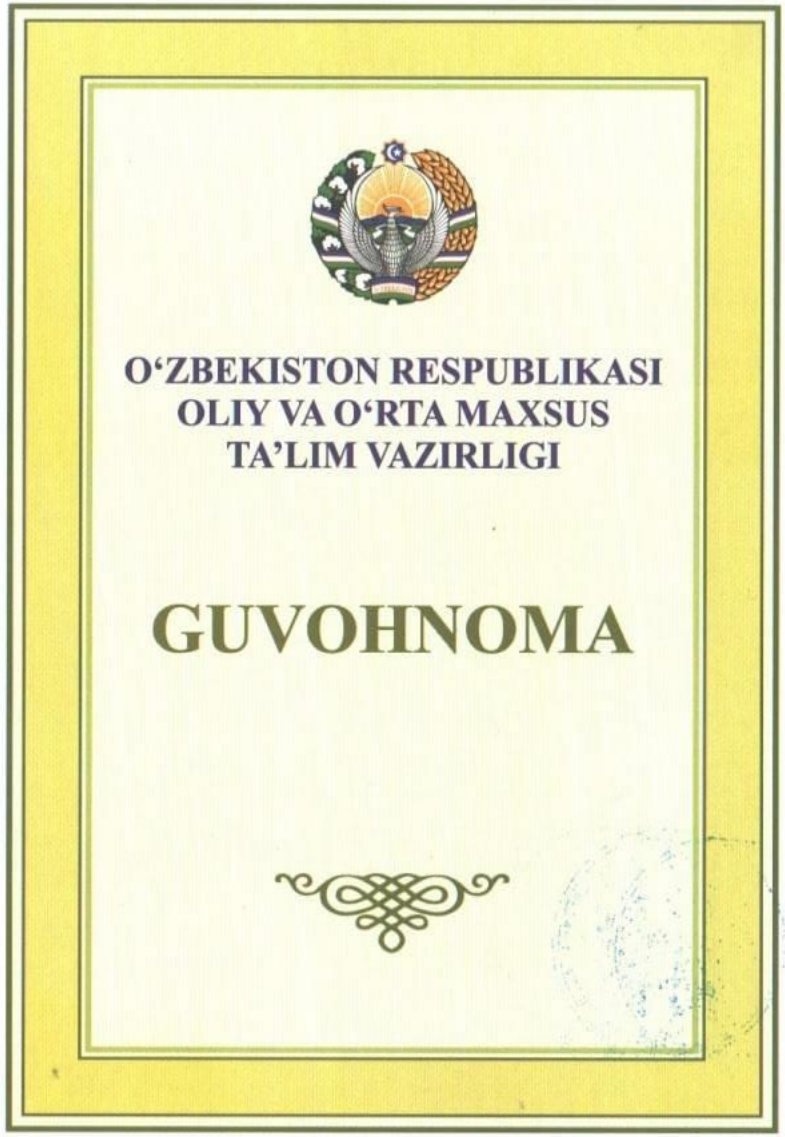
Все данные - пользователи и их сообщения будут храниться в переменной chatModel. И так как она будет общим для всех пользователей, она объявлена статической.
Метод Index принимает четыре параметра. Первый параметр идентифицирует пользователя. Параметр logOn при значении true указывает, что пользователь осуществил вход. Параметр logOff при значении true, наоборот, указывает, что пользователь вышел из чата. И последний параметр содержит сообщение пользователя. И в зависимости от значений параметров возвращаем либо обычное представление Index.cshtml, либо одно из частичных представлений.
Если запрос не является AJAX-запросом, передаеся обычное представление. Если установлен параметр logOn или logOff, то возвращаем частичное представление ChatRoom. И если установлен последний параметр с сообщением пользователя? то возвращается частичное представление History
Теперь определим все эти представления.

Заключение

Предмет программной инженерии учит синтаксису программной инженерии и работе на необходимых платформах для создания практических программ на нем, для создания программного обеспечения произвольной сложности, для создания современного распределенного программного обеспечения. Далее в рамках этого курса студенты изучают технологии программирования для работы MVC 5 и проектами в них и получают возможность эффективно использовать их в практической работе, исследованиях, а также в системе образования.

Использованная литература

1. Roger Pressman, Bruce Maxim, Software Engineering: A Practitioner's Approach, John Wiley & Sons, USA 2014.
2. Ian Sommerville. Software Engineering Hardcover. Pearson 2010 USA
3. Robert W. Sebesta, Concepts of Programming Languages, John Wiley & Sons, USA 2015.
4. Fundamentals of Computer Programming With C# (The Bulgarian C# Programming Book). Svetlin Nakov & Co., 2013.
5. Шилдт, Герберт. C# 4.0: полное руководство. : Пер. с англ. — М. : ООО "И.Д. Вильяме", 2011.



O‘ZBEKISTON RESPUBLIKASI
OLIY VA O‘RTA MAXSUS
TA’LIM VAZIRLIGI

GUVOHNOMA

O‘QUV ADABIYOTINING NASHR RUXSATNOMASI

O‘zbekiston Respublikasi Oliy va o‘rta maxsus ta’lim vazirligining 2022 yil “19” iyul dagi “233” -sonli buyrug‘iga asosan

Д.П.Мухамедиева, А.П.Варламова, С.А.Бахромов,
(muallifning familiyasi, ismi-sharifi)
Б.Б.Элов, О.Н.Абдурахмонов

5330100 - Axborot tizimlarining matematik va dasturiy ta’minoti
(ta’lim yo‘nalishi (mutaxassisligi))

ning

talabalari (o‘quvchilari) uchun tavsiya etilgan

Программная инженерия ASP.NET MVC – 5.0 nomli o‘quv
(o‘quv adabiyotining nomi va turi: darslik, o‘quv qo‘llanma)
qo‘llanmasi

ga

O‘zbekiston Respublikasi Vazirlar Mahkamasi tomonidan litsenziya berilgan nashriyotlarda nashr etishga ruxsat berildi.

M.O‘

Vazir _____ A.Toshkulov
(imzo)

Ro‘yxatga olish raqami
233-0797