ВВЕДЕНИЕ В ASP.NET MVC 5

1 Создание первого приложения на ASP.NET MVC 5

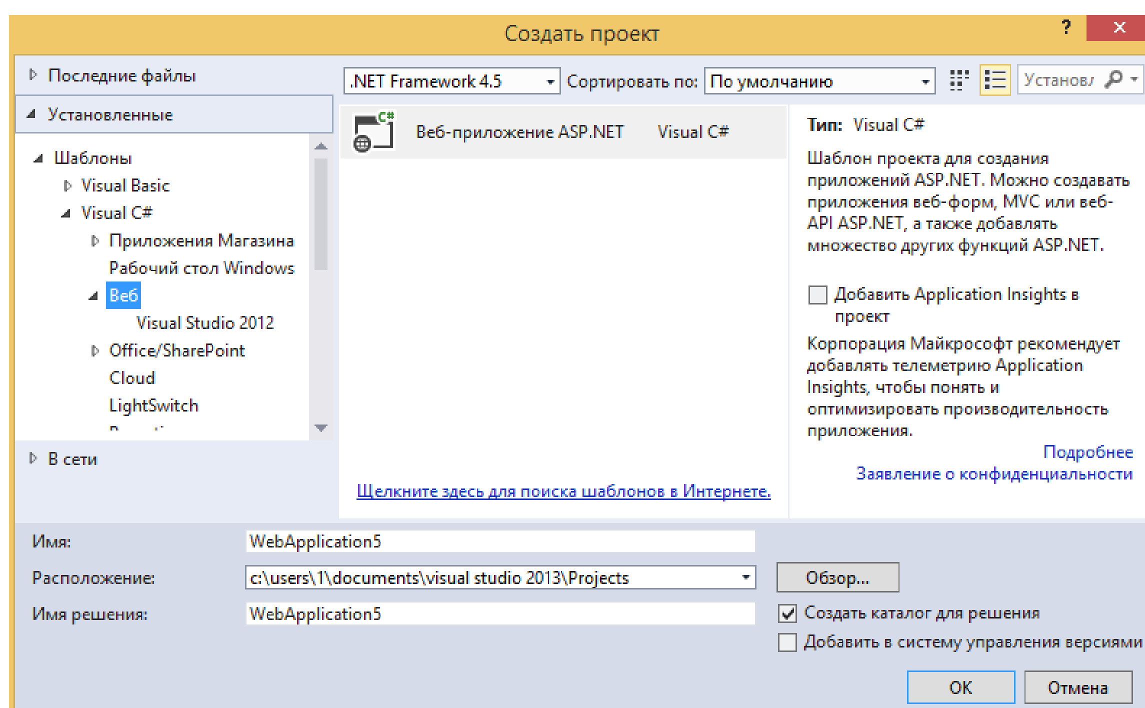
1.1 Создание проекта.

Создадим первое приложение. Это будет очень простенькое приложение, цель которого - дать некоторое начальное понимание работы с ASP.NET MVC 5.

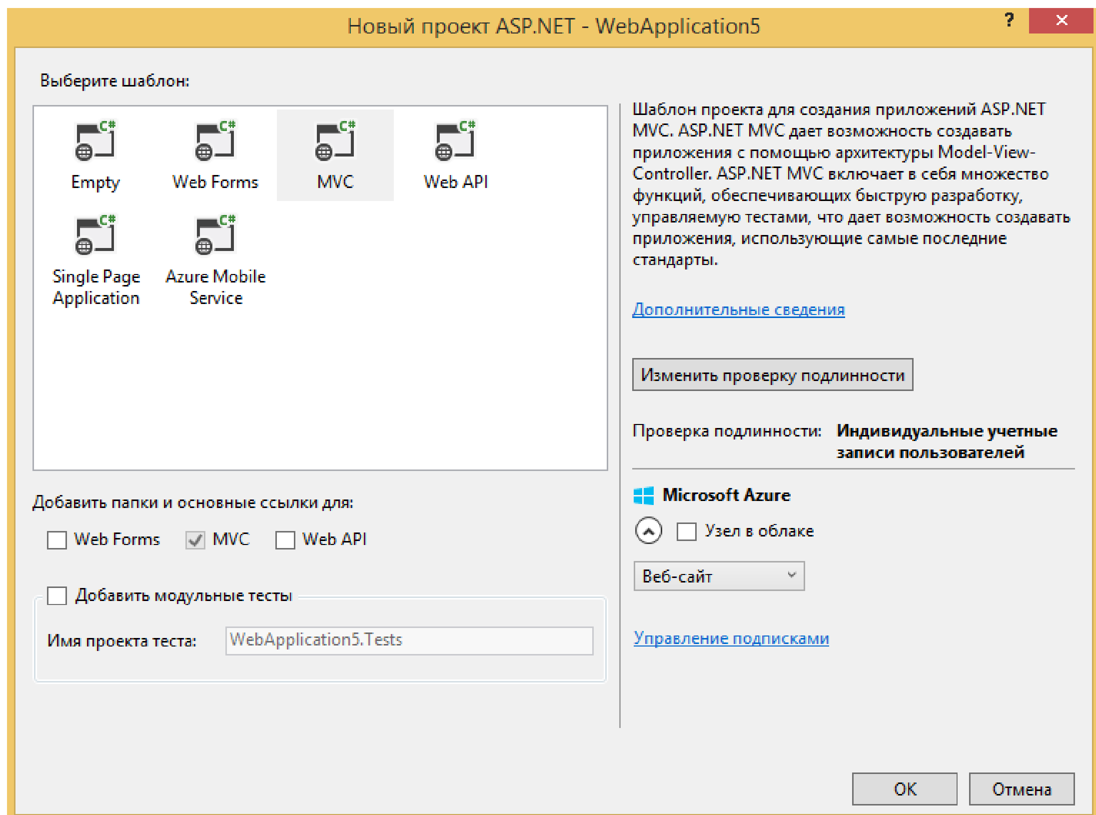
Это приложение будет эмулировать работу книжного магазина: оно будет **предоставлять нам выбор книг**, а пользователь, зашедший на сайт, сможет **оформить покупку**. Для начала, я думаю, достаточно.

Для создания веб-приложений на платформе ASP.NET MVC 5 необходима среда разработки - Visual Studio Community 2013 (либо другой выпуск Visual Studio 2013), которую можно найти по адресу Visual Studio Community 2013.

После установки откроем Visual Studio 2013 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.

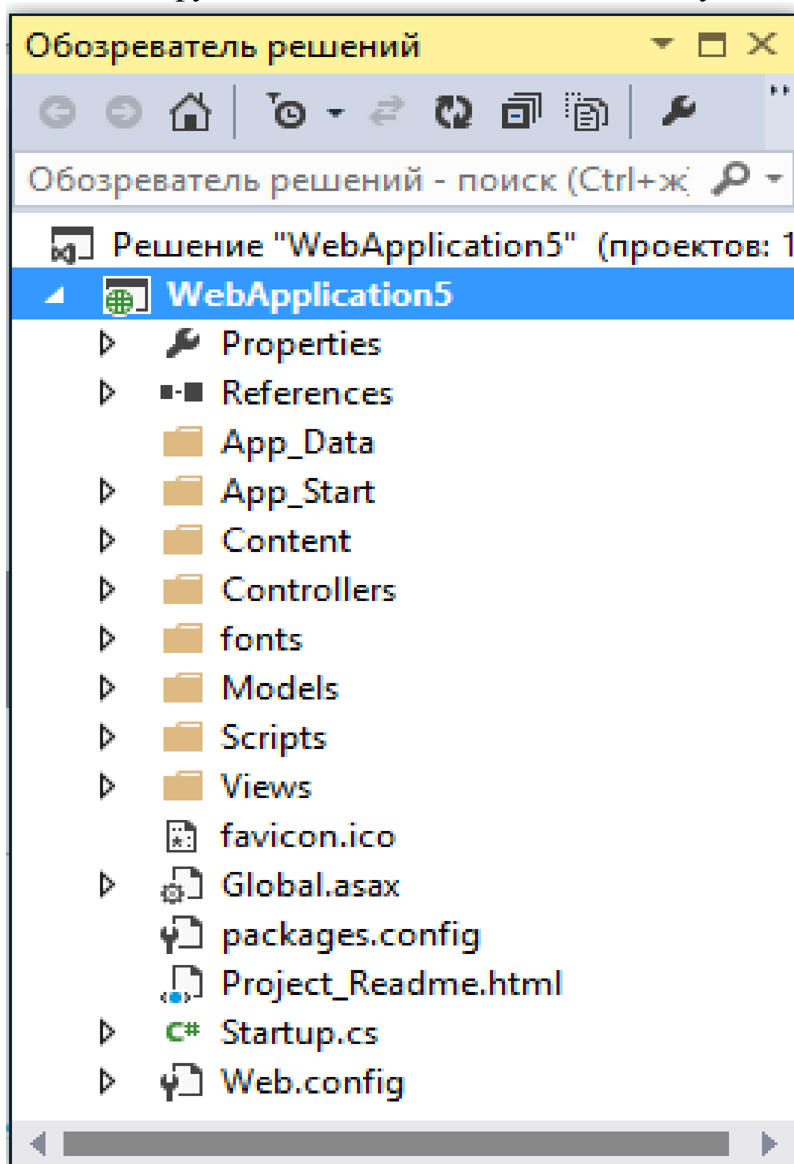


Также нам доступен в правой части окна выбор механизма аутентификации в приложении (кнопка **Change Authentication**). По умолчанию установлен тип **Individual User Accounts**. Не будем его изменять.

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:

1.2 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.

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 WebApplication5.Models
{   public class Book
    {
        // ID книги
        public int Id { get; set; }
        // название книги
        public string Name { get; set; }
        // автор книги
        public string Author { get; set; }
        // цена
        public int Price { get; set; }
    }
}
```

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

```
namespace WebApplication5.Models
{
   public class Purchase
   {
      // ID покупки
      public int PurchaseId { get; set; }
      // имя и фамилия покупателя
      public string Person { get; set; }
      // адрес покупателя
      public string Address { get; set; }
      // ID книги
      public int BookId { get; set; }
      // дата покупки
      public DateTime Date { get; set; }
   }
}
```

1.3 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.

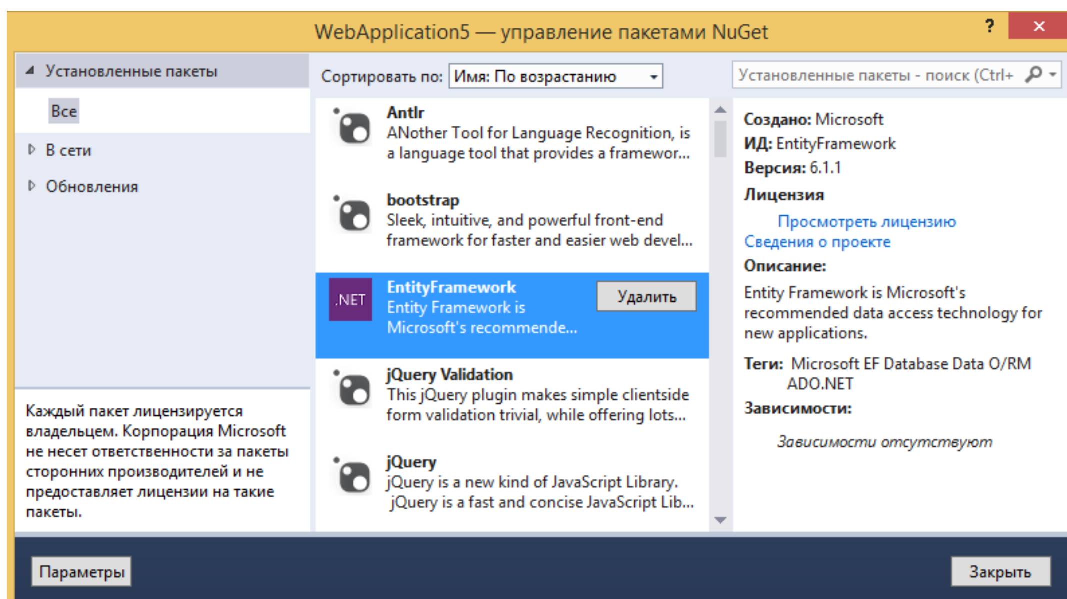
Второй способ состоял в определении ключа с помощью атрибута Key, установленным над нужным свойством.

1.4 EntityFramework

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 EntityFramework. 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 **EntityFramework** 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 EntityFramework, который нам и надо установить:



Запустим процесс установки пакета, нажав на кнопку Install.

1.5 Создание контекста данных

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

```
namespace WebApplication5.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).

CodeFirst

Хотя мы будем использовать базу данных, но создавать явным образом мы ее не будем. За нас все сделает EntityFramework. 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*:

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

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.

При создании подключения надо учитывать версию MS SQL Servera, с которой предстоит работать. Выше приведена строка подключения для MS SQL Server 2012. При использовании версии MS SQL Server 2014 строка подключения может немного отличаться:

1.6 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:

```
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 System.Data.Entity;
using WebApplication5.Models;

namespace WebApplication5.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();
        }

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

            return View();
        }

        public ActionResult Contact()
        {
```

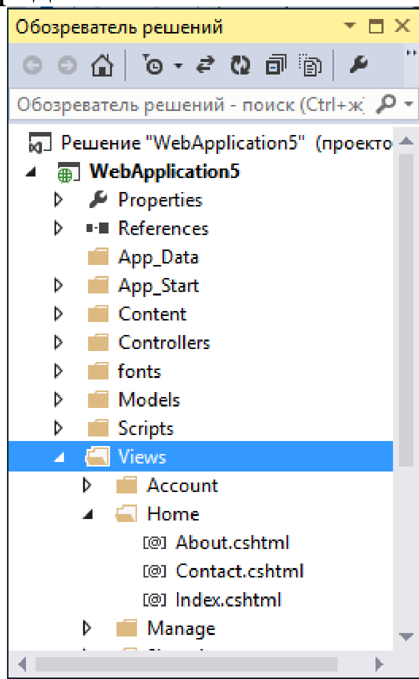
```
            ViewBag.Message = "Your contact page.";

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

Для передачи списка объектов Book в представление 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*.



Откроем представление **Index.cshtml** 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 Купить. 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 Buy, но скоро мы его создадим.

1.7 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.

1.8 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 =
```

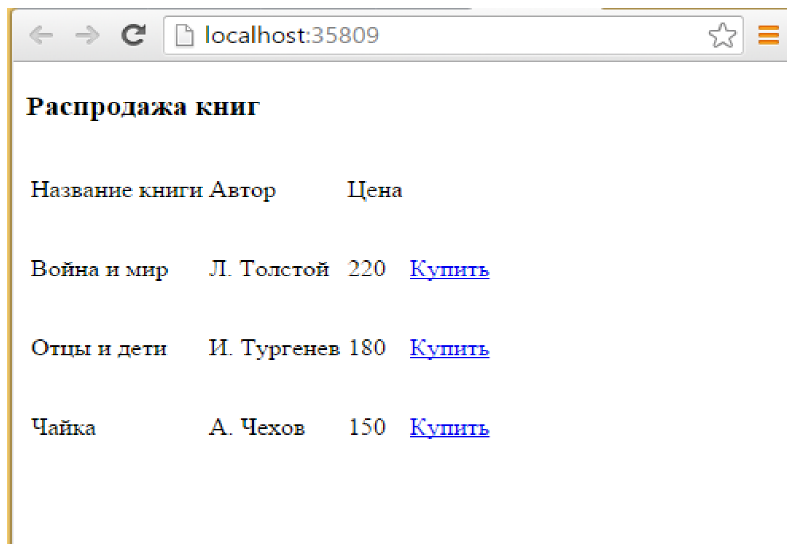
```
150 });

            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.Web.Routing;
using WebApplication5.Models;
using System.Data.Entity;

namespace WebApplication5
{
    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 на жестком диске и в этой папке перейдем к каталогу **App_Data**, то сможем 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

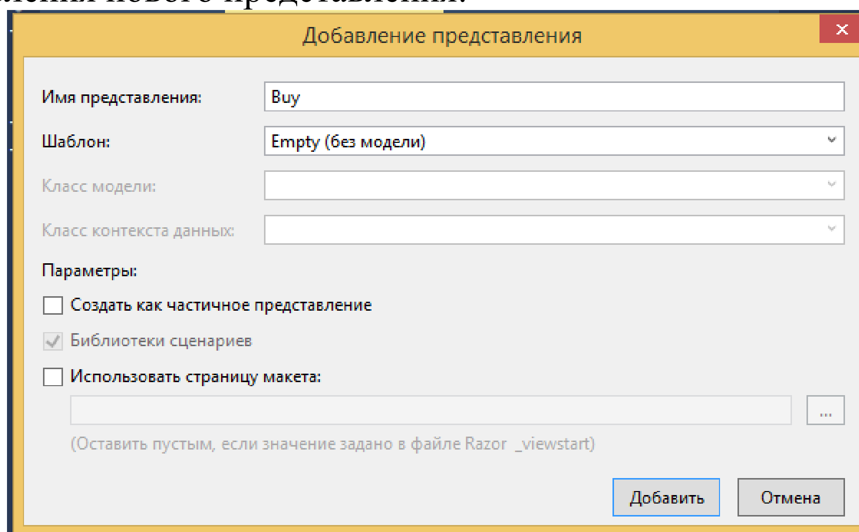
параметр передается через объект ViewBag в представление, которое мы сейчас создадим.

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 EntityFramework 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:

```
<!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>
```

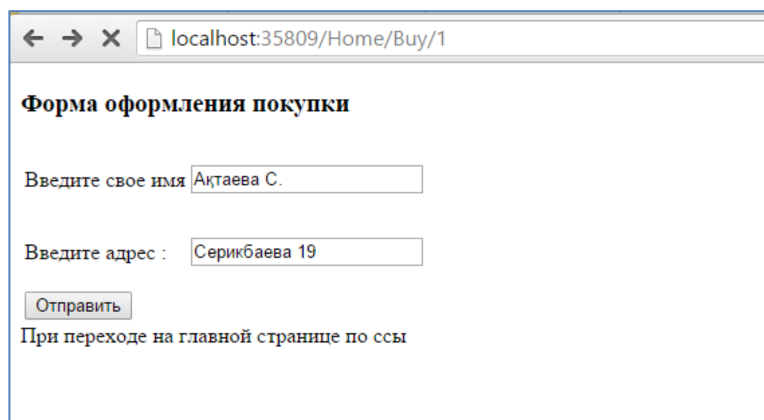
При переходе на главной странице по ссылке "/Home/Buy/2" контроллер будет получать запрос к действию Buy, передавая ему в качестве параметра id значение 2. И так как такой запрос представляет тип GET, пользователю будет возвращаться данное представление с формой.

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.

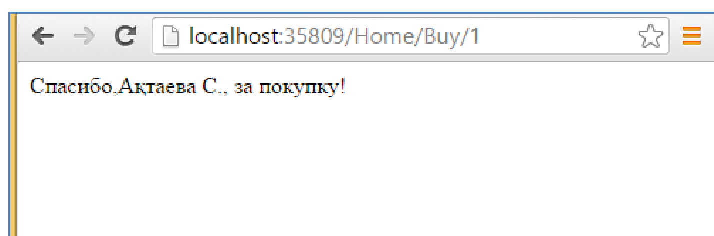
После заполнения формы и нажатия на кнопку форма будет оправляться запросом POST, так как мы его определили в строке <form method="post" action="">. Контроллер снова будет получать запрос к методу Buy, только теперь будет выбираться для обработки запроса метод public string Buy(Purchase purchase).

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:



2 Контроллеры

2.1 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*.

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


namespace WebApplication5.Controllers
{
    public class MyController : IController
    {
        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.

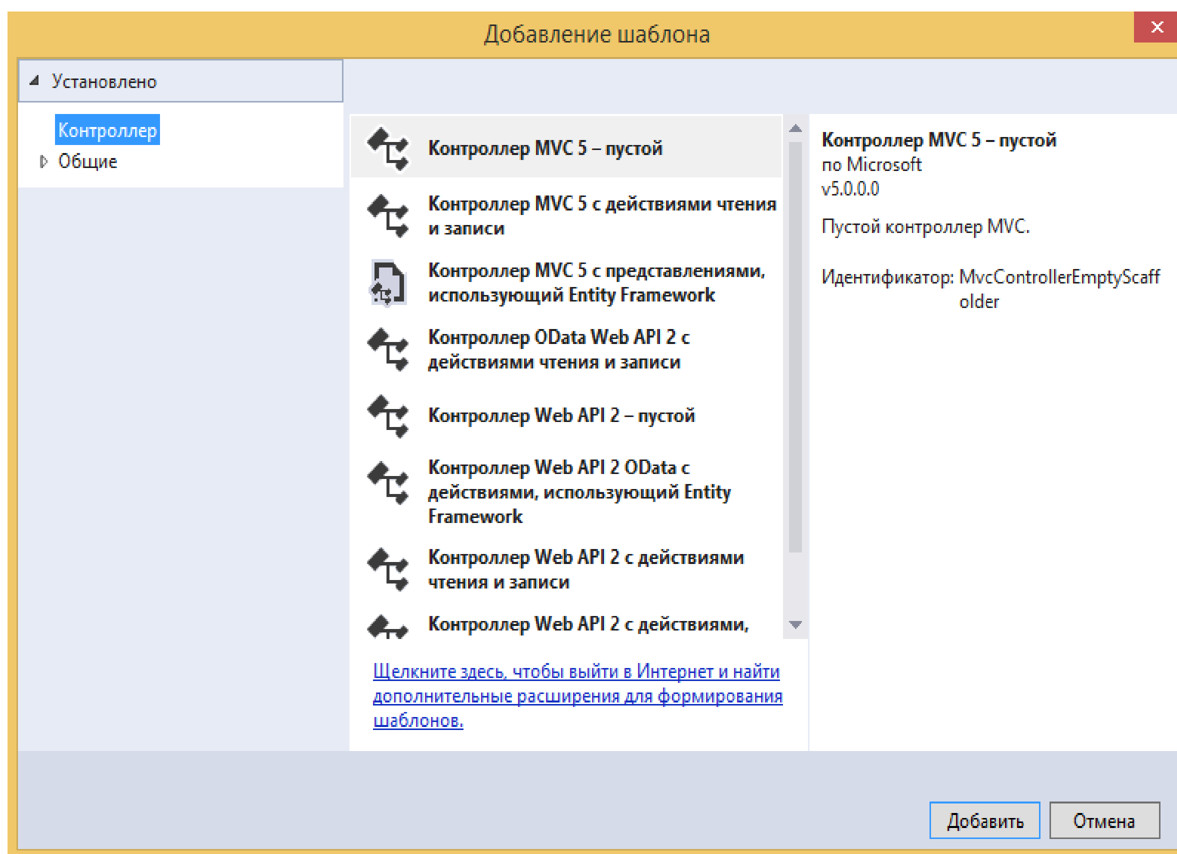
Кроме того, мы можем отправить пользователю ответ с помощью объекта **Response** и его метода Write.

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.

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:

```
public class HomeController : Controller
{
   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);
}
```

Встроенные классы, производные от ActionResult

В реальности нам вряд ли потребуется часто создавать свои классы для обработки результата действия. Фрейморк **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
- 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

3 Представления

3.1 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 словоdynamic. 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:

Добавление представления

Имя представления: View

Шаблон: Empty (без модели)

Класс модели:

Класс контекста данных:

Параметры:

Создать как частичное представление

Библиотеки сценариев

Использовать страницу макета:

(Оставить пустым, если значение задано в файле Razor _viewstart)

Добавить Отмена

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 отображает все объекты из списка моделей в виде таблицы. 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 (BookStore.Models.Book b in Model)
{
   <p>@b.Name</p>
}
```

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">Купить</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:

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

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

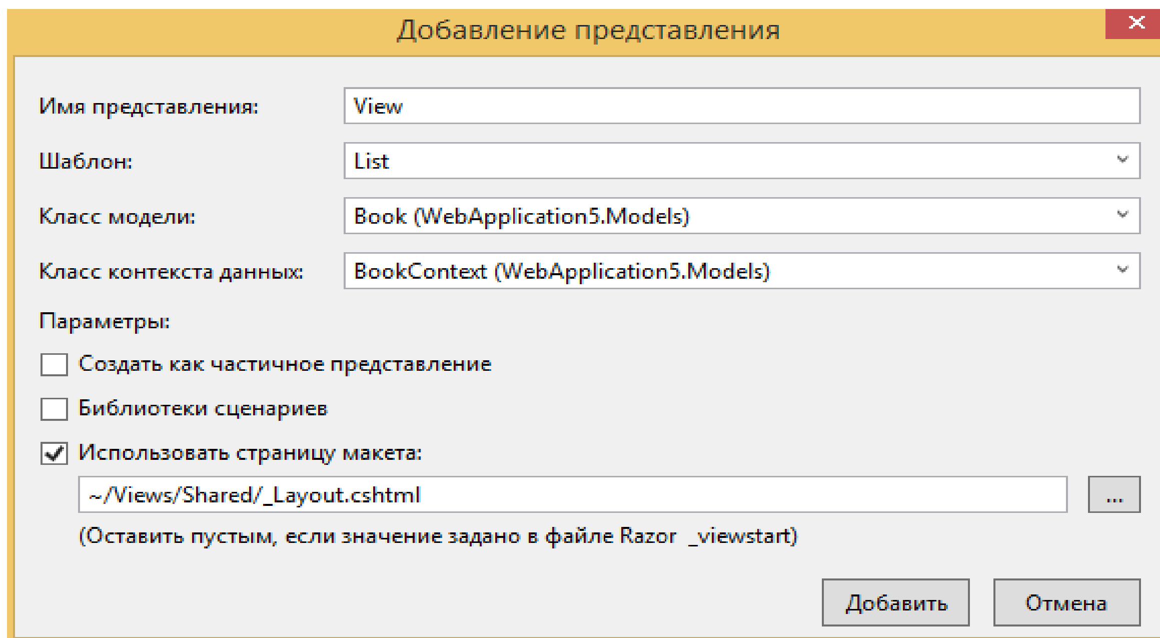
```
@model IEnumerable<BookStore.Models.Book>
@{
   Layout = "~/Views/Shared/_Layout.cshtml";
}
   <div>
      <h3>Распродажа книг</h3>
      <table>
         <tr class="header"><td><p>Название книги</p></td>
            <td><p>Автор</p></td>
            <td><p>Цена</p></td><td></td>
         </tr>
         @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>
```

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

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.

4 Задания для выполнения

1. Ознакомится с материалами, изложенных в методическом указаний.
2. Определить тематику веб-приложения;
3. Добавить модель объектов в структуре проекта (папка Models);
4. Добавить контекст данных для работы с данными;
5. В структуре проекта добавить контроллеры с методами
6. Создать несколько представлений используя синтаксиса Razor.