«Учебная практика №2»

Оглавление

1. 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.

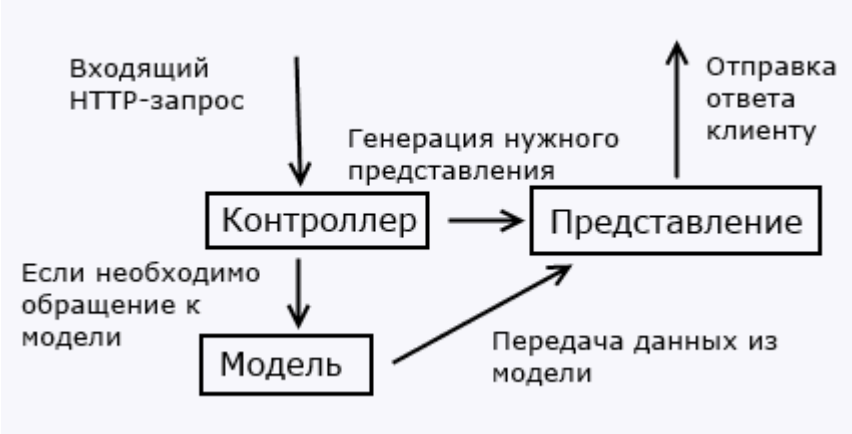
Концепция паттерна (шаблона) MVC (model - view - controller) предполагает разделение приложения на три компонента:

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.

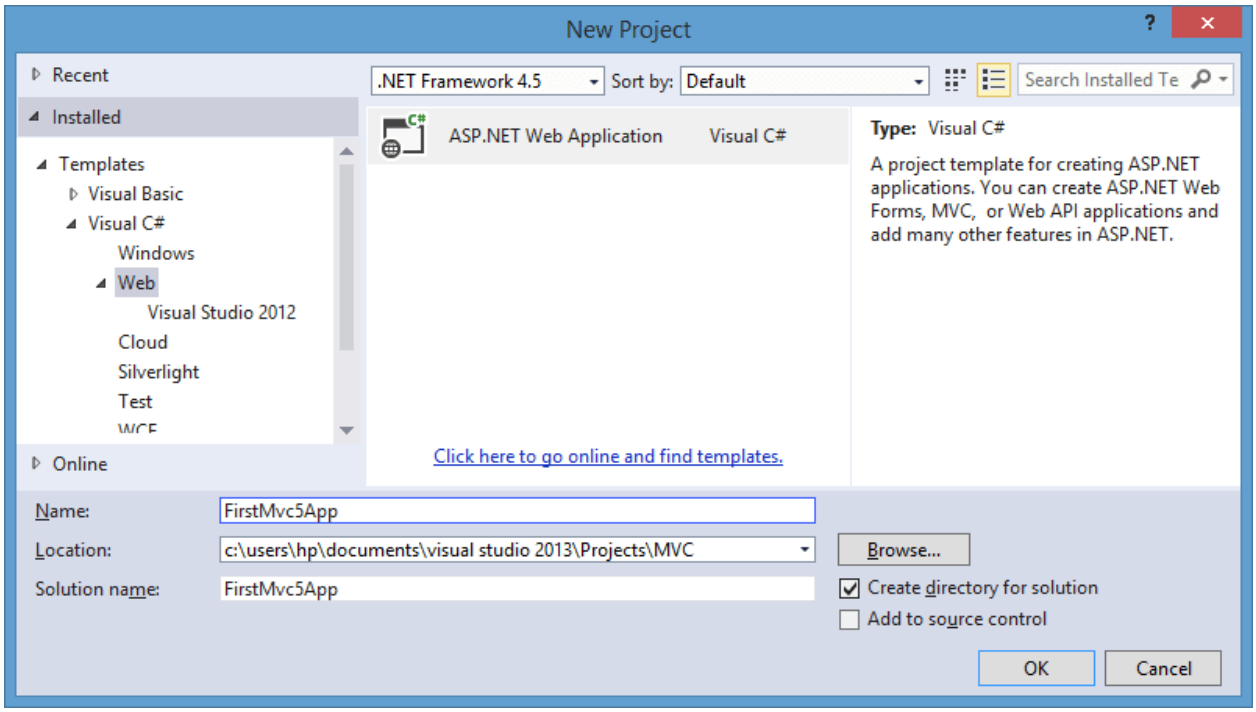
Конкретные реализации и определения данного паттерна могут отличаться, но в силу своей гибкости и простоты он стал очень популярным в последнее время, особенно в сфере веб-разработки.

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.

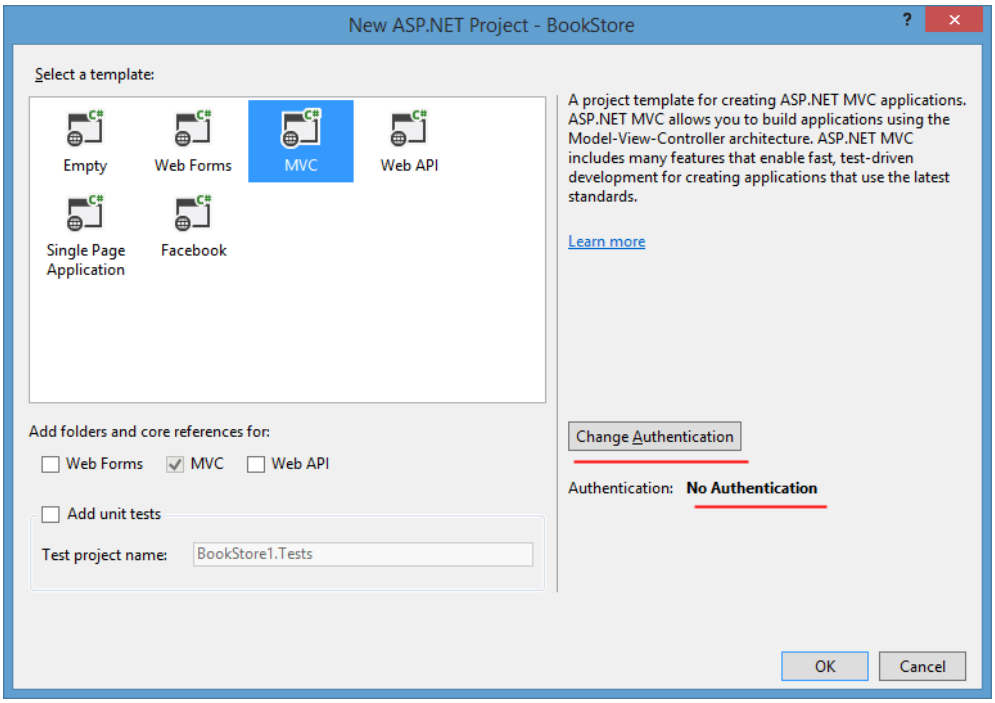
2. Начало работы с ASP.NET MVC 5

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:

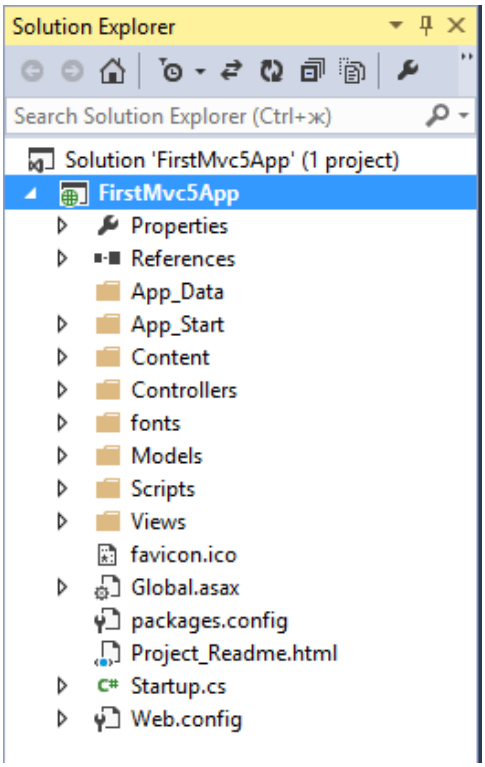


По умолчанию уже выбран шаблон MVC. А в правой части окна изменим тип аутентификации приложения на **No Authentication** (так как пока нам система аутентификации не нужна):

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

2.1. 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.

3. 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. Модели представляют собой простые классы и располагаются в проекте в каталоге Models. Модели описывают логику данных. Например, модель, представляющая футболистов и футбольную команду:

```
namespace FootballExample.Models
{
    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; }
    }
}

namespace FootballExample.Models
{
    public class Team
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Coach { get; set; }

        public ICollection<Player> Players { get; set; }
        public Team()
        {
            Players = new List<Player>();
        }
    }
}
```

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.

3.1. 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 Player определено свойство Id, то есть данное свойство является первичным ключом.

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

3.2. 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.

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.

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.

Если при создании проекта MVC 5 вы выберете в качестве типа аутентификации "No Authentication", то после создания проекта в его надо будет подключить EntityFramework через пакетный менеджер NuGet.

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 EntityFramework
```

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:

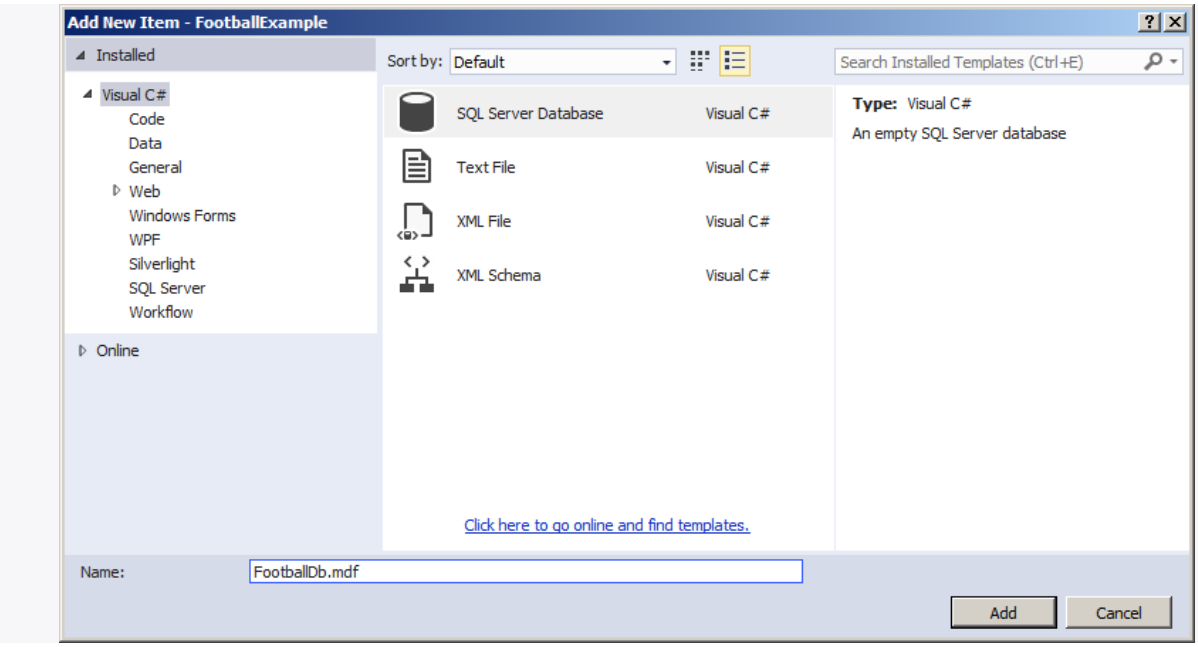
```
namespace FootballExample.Models
{
    public class FootballContext : DbContext
    {
        public FootballContext()
            : base("FootballEntities")
        {

        }

        public DbSet<Player> Players { get; set; }
        public DbSet<Team> Teams { get; set; }

    }
}
```

С помощью свойств Players и Teams мы получаем доступ к данным соответствующих моделей, которые хранятся в базе данных.

3.3. 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 FootballDb.mdf:



После этого в папку папку App_Data будет добавлена база данных, и мы можем начинать с ней работать - добавлять таблицы и данные.

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="FootballEntities"
         connectionString="Data Source=(LocalDB)\v11.0; AttachDbFilename='|DataDirec-
tory|\FootballDb.mdf'; Integrated Security=True"
         providerName="System.Data.SqlClient"/>
  </connectionStrings>
```

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.

3.4. Данные для моделей по умолчанию

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

```
namespace FootballExample.Models
{
    public class FootballDbInitializer
        : DropCreateDatabaseAlways<FootballContext>
    {
        protected override void Seed(FootballContext db)
        {
            // создаем футболистов
            var player1 = new Player
            {
                Name = "Лионель Месси",
```

```
            Age = 28,
            Position = "Нападающий"
        };

        var player2 = new Player
        {
            Name = "Криштиану Роналду",
            Age = 30,
            Position = "Нападающий"
        };
        var player3 = new Player
        {
            Name = "Неймар",
            Age = 23,
            Position = "Нападающий"
        };
        var player4 = new Player
        {
            Name = "Луис Суарес",
            Age = 28,
            Position = "Нападающий"
        };
        var player5 = new Player
        {
            Name = "Серхио Агуэро",
            Age = 27,
            Position = "Нападающий"
        };

        db.Players.Add(player1);
        db.Players.Add(player2);
        db.Players.Add(player3);
        db.Players.Add(player4);
        db.Players.Add(player5);

        // создаем команды
        var team1 = new Team { Name = "Барселона", Coach = "Луис Энрике" };
        team1.Players.Add(player1);
        team1.Players.Add(player3);
        team1.Players.Add(player4);

        var team2 = new Team { Name = "Реал Мадрид", Coach = "Рафаэль Бенитес" };
        team2.Players.Add(player2);

        var team3 = new Team { Name = "Манчестер Сити", Coach = "Мануэль Пеллегрини"
};
        team3.Players.Add(player5);

        db.Teams.Add(team1);
        db.Teams.Add(team2);
        db.Teams.Add(team3);

        base.Seed(db);
    }
}

}
```

Класс DropCreateDatabaseIfModelChanges позволяет при каждом изменении модели удалять созданную БД и создавать ее заново, при этом заполняя её некоторыми начальными данными. В качестве таких начальных значений здесь создаются несколько футболистов и

команд. Используя метод db.Players.Add и db.Teams.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 FootballDbInitializer());:

```
namespace FootballExample
{
    public class MvcApplication : System.Web.HttpApplication
    {
        protected void Application_Start()
        {
            Database.SetInitializer(new FootballDbInitializer());
            AreaRegistration.RegisterAllAreas();
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);
        }
    }
}
```

4. Контроллеры. Основы контроллеров

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:

```
namespace FootballExample.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();
        }
    }
}
```

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/Show - обращение к методу Show 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. Итак, добавим в папку Controllers проекта новый класс (именно класс, а не контроллер) со следующим содержанием:

```
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.

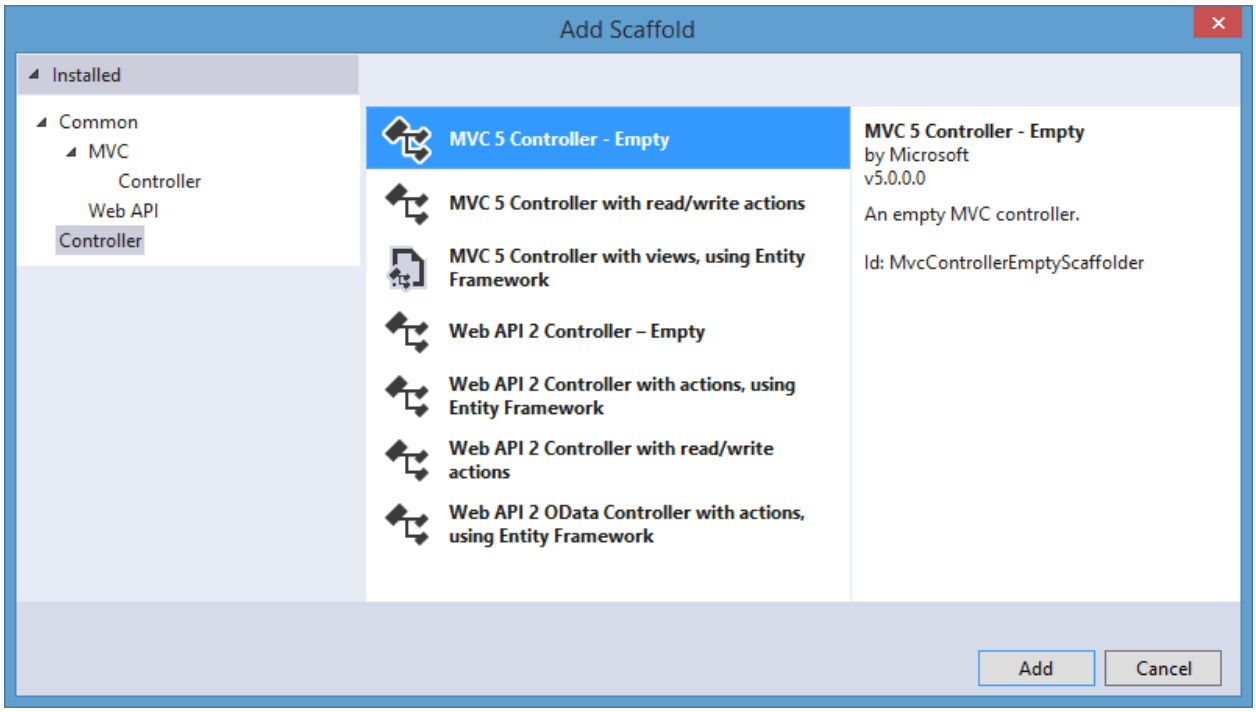
Хотя с помощью реализации интерфейса IController очень просто создавать контроллеры, но в реальности чаще оперируют более высокоуровневыми классами, как например класс Controller, поскольку он предоставляет более мощные средства для обработки запросов. И если при реализации интерфейса IController мы 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.

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

```
namespace FootballExample.Controllers
{
    public class FootballController : Controller
    {
        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 позже.

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.

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

```
namespace FootballExample.Controllers
{
    public class HomeController : Controller
    {
        // создаем контекст данных
        private readonly FootballContext _db = new FootballContext();
        public ActionResult Index()
        {
            // получаем из бд все объекты Book
            var players = _db.Players.ToList();
            // передаем все объекты в динамическое свойство Players в ViewBag
            ViewBag.Players = players;
            // возвращаем представление
            return View();
        }
    }
}
```

Прежде всего, мы подключаем пространство имен моделей, даже несмотря на то, что он находится в одном проекте, но в разных пространствах. Затем создается объект контекста данных, через который мы будем взаимодействовать с бд:

```
FootballContext _db = new FootballContext();
```

Далее используя метод _db.Players, получаем из базы данных набор объектов Player.

Для передачи списка объектов Player 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.Players, которая и будет хранить набор игроков.

5. Представления

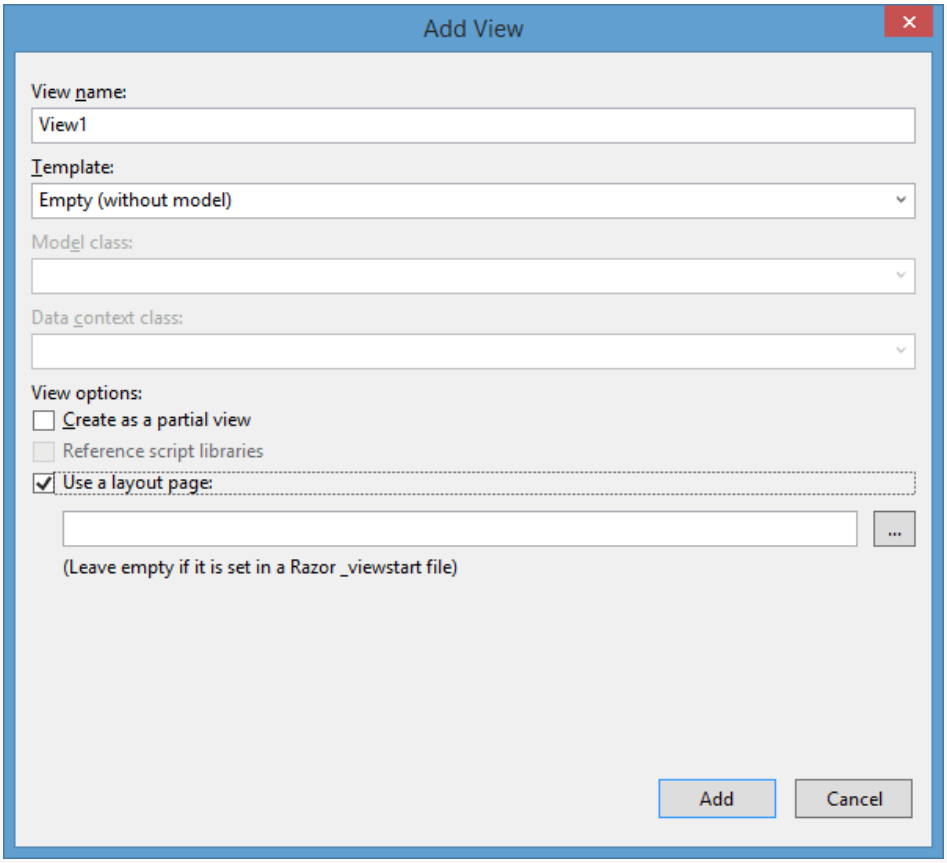
Хотя работа приложения MVC управляется главным образом контроллерами, но непосредственно пользователю приложение доступно в виде представления, которое и формирует внешний вид приложения. В ASP.NET MVC 5 представления - это файлы с расширением cshtml, которые содержат код пользовательского интерфейса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

5.1. 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.

- 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 до-

ступным 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.

5.2. 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()
{
    var players = _db.Players.ToList();
    ViewBag.Players = players;
    return View();
}
```

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

```
public ActionResult Index()
{
    var players = _db.Players.ToList();
    ViewBag.Players = players;
    return View("~/Views/Some/SomeView.cshtml");
}
```

5.3. 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.

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#.

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>
```

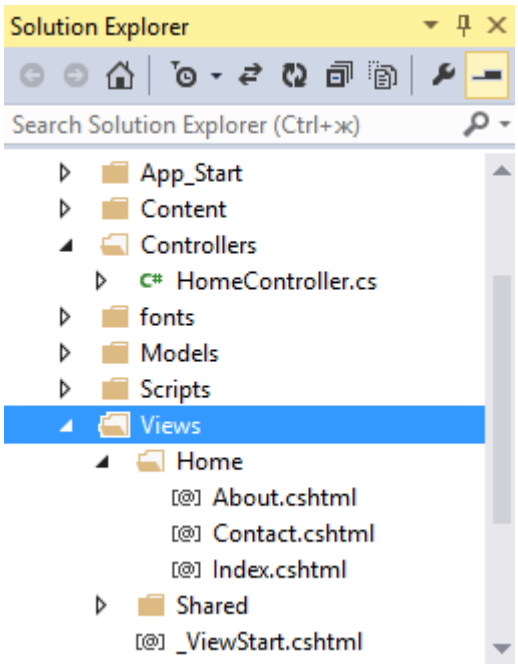
Применение блоков кода аналогично, только знак @ ставится перед всем блоком кода, а движок автоматически определяет, где этот блок кода заканчивается:

```
@foreach (var player in ViewBag.Players)
{
    <p>@player.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>
```

Теперь создадим само представление для вывода списка игроков. 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>Футболисты</title>
</head>
<body>
    <div>
        <h3>Список игроков</h3>
        <table>
            <tr>
                <td><p>Имя</p></td>
                <td><p>Возраст</p></td>
                <td><p>Позиция</p></td>
                <td></td>
            </tr>
            @foreach (var p in ViewBag.Players)
            {
                <tr>
                    <td><p>@p.Name</p></td>
                    <td><p>@p.Age</p></td>
                    <td><p>@p.Position</p></td>
                    <td><p><a href="/Home/Vote/@p.Id">Голосовать</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 футболистах. Здесь также используется интересная конструкция @foreach (var p in ViewBag.Players). То есть тут мы создаем цикл. В нем мы пробегаемся по всем элементам в объекте ViewBag.Players, 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 Голосовать. При нажатии на эту ссылку методу Vote контроллера HomeController будет отправляться запрос, в котором вместо @p.Id будет указан id игрока. Пока у нас, правда, отсутствует метод Vote, сейчас мы его создадим.

Метод Vote, отвечает за голосование за игрока по десятибалльной шкале. Добавим в контроллер HomeController следующие два метода:

```
[HttpGet]
public ActionResult Vote(int id)
{
    var player = _db.Players.Find(id);

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

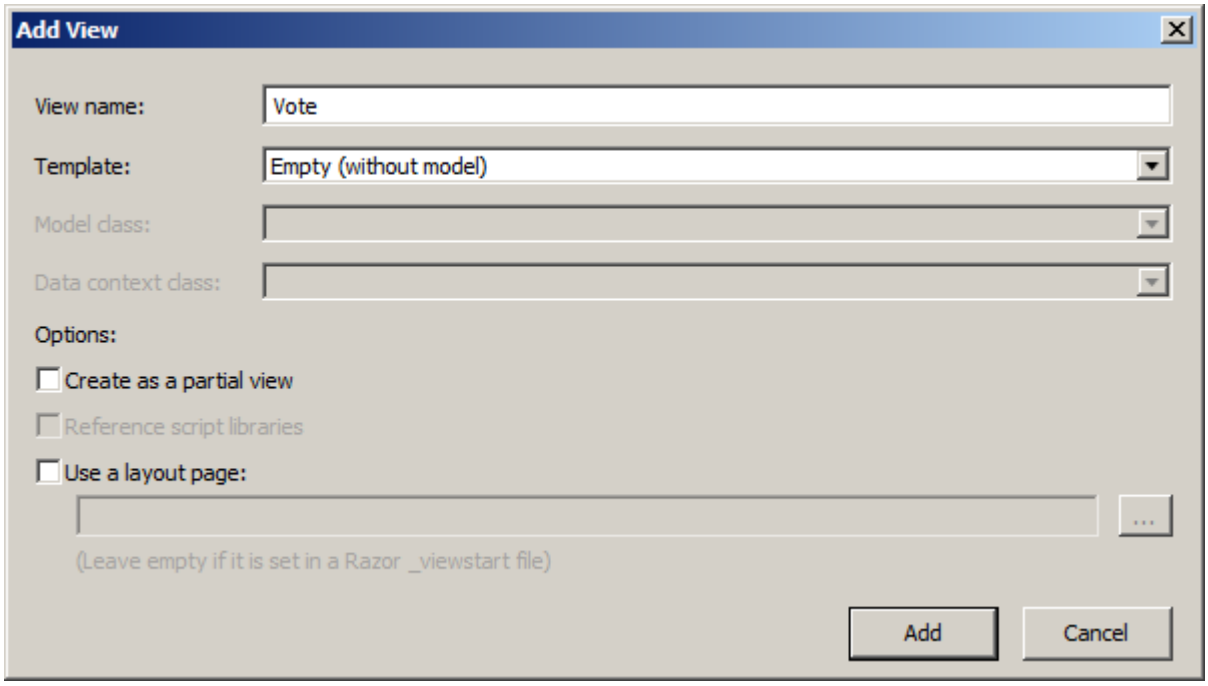
```
            ViewBag.Name = player.Name;

            return View();
        }

        [HttpPost]
        public string Vote(string name)
        {
            return "Спасибо, " + name + ", за ваш голос!";
        }
```

Хотя здесь два метода, но в целом они составляют одно действие Vote, 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 Vote будет передаваться id игрока, за которого пользователь хочет проголосовать, то нам надо определить в методе соответствующий параметр: public ActionResult Vote(int id). Затем этот параметр передается через объект ViewBag в представление, которое мы сейчас создадим.

Метод public string Vote(string name) принимает переданную ему в запросе POST строку с именем пользователя и выводит сообщение на экран.

И в конце добавим представление Vote.cshtml. Для этого нажмем на метод public ActionResult Vote(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>
```

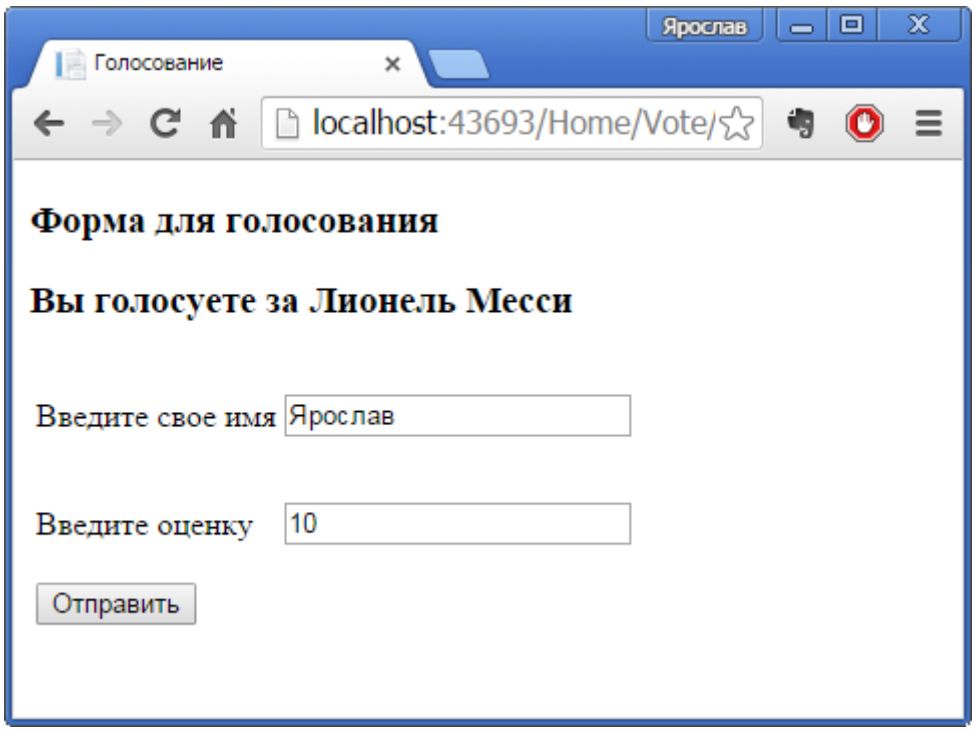
```
    <meta name="viewport" content="width=device-width" />
    <title>Голосование</title>
</head>
<body>
    <div>
        <h3>Форма для голосования</h3>
        <form method="post" action="">
            <h3>Вы голосуете за @ViewBag.Name</h3>
            <table>
                <tr>
                    <td><p>Введите свое имя </p></td>
                    <td><input type="text" name="Name" /> </td>
                </tr>
                <tr>
                    <td><p>Введите оценку</p></td>
                    <td>
                        <input type="text" name="Score" />
                    </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/Vote/2" контроллер будет получать запрос к действию Vote, передавая ему в качестве параметра id значение 2. И так как такой запрос представляет тип GET, пользователю будет возвращаться данное представление с формой.

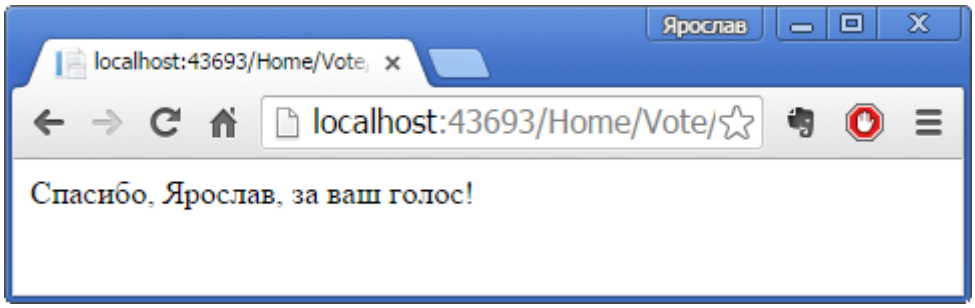
Представление по сути представляет собой форму для ввода данных. После заполнения формы и нажатия на кнопку форма будет оправляться запросом POST, так как мы его определили в строке <form method="post" action="">. Контроллер снова будет получать запрос к методу Vote, только теперь будет выбираться для обработки запроса метод public string Vote(string Name).

Как система MVC угадывает, что мы передали с запросом post информацию об имени игрока? Обратите внимание на поле ввода <input type="text" name="Name" />. Значение атрибута name соответствует имени аргумента метода.

Теперь запустим наше приложение. На главной странице выберем какого-нибудь игрока и нажмем на ссылку "Голосовать". На форме заполним поля и нажмем кнопку "Отправить".



После нажатия кнопки браузер отобразит уведомление:



6. Методы действий и их параметры

Методы действий (action methods) представляют такие методы контроллера, которые обрабатывают запросы по определенному URL. 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 Vote 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,

```
[HttpGet]
public ActionResult Vote(int id)
{
    var player = _db.Players.Find(id);

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

```
        ViewBag.Name = player.Name;
        ViewBag.Date = GetToday();

        return View();
    }

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

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

6.1. Передача данных в контроллеры и параметры

В рассмотренном нами приложении из метод Vote использовал параметр name.

Значение атрибута name у одного из полей ввода должно соответствовать имени аргумента, поэтому система автоматически свяжет значения полей с соответствующими аргументами

Кроме POST-запросов у нас есть также GET-запросы, при которых все параметры передаются в строке запроса. Например, вторая версия метода Vote в качестве параметра принимает значение типа int: public ActionResult Vote(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/Vote?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/Vote/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).

6.2. 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>";
        }
```

7. 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 предыдущем разделе 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 проект, из прошлого раздела, и добавим в него новую папку Util, которая будет содержать новые классы. После добавления папки добавим в нее первый класс. Назовем его HtmlResult. Он у нас будет содержать следующий код:

```
namespace FootballExample.Util
{
    public class HtmlResult : ActionResult
    {
        private readonly 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:

```
namespace FootballExample.Util
{
    public class ImageResult : ActionResult
    {
        private readonly 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:cen-
ter;'>" +
                "<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/cat.png";
    return new ImageResult(path);
}
```

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

7.1. 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, 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.

7.2. 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 ActionResult Index()
{
       ViewBag.Players = players;
       return View("~/Views/Some/SomeView.cshtml");
}
```

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

7.3. 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:

```
.........................
<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()
{
    var players = _db.Players.ToList();
    ViewBag.Players = players;
    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.

8. 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:

```
[HttpGet]
public ActionResult Vote(int id)
{
    var player = _db.Players.Find(id);

    if (player == null)
    {
        return Redirect("Home/Index");
    }

    ViewBag.Name = player.Name;

    return View();
}
```

Если в качестве параметра будет передан идентификатор несуществующего игрока, 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.

8.1. 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 Check(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 Check(int age)
{
    if (age < 21)
    {
        return new HttpUnauthorizedResult();
    }
    return View();
}
```

9. Строго типизированные представления

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

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:

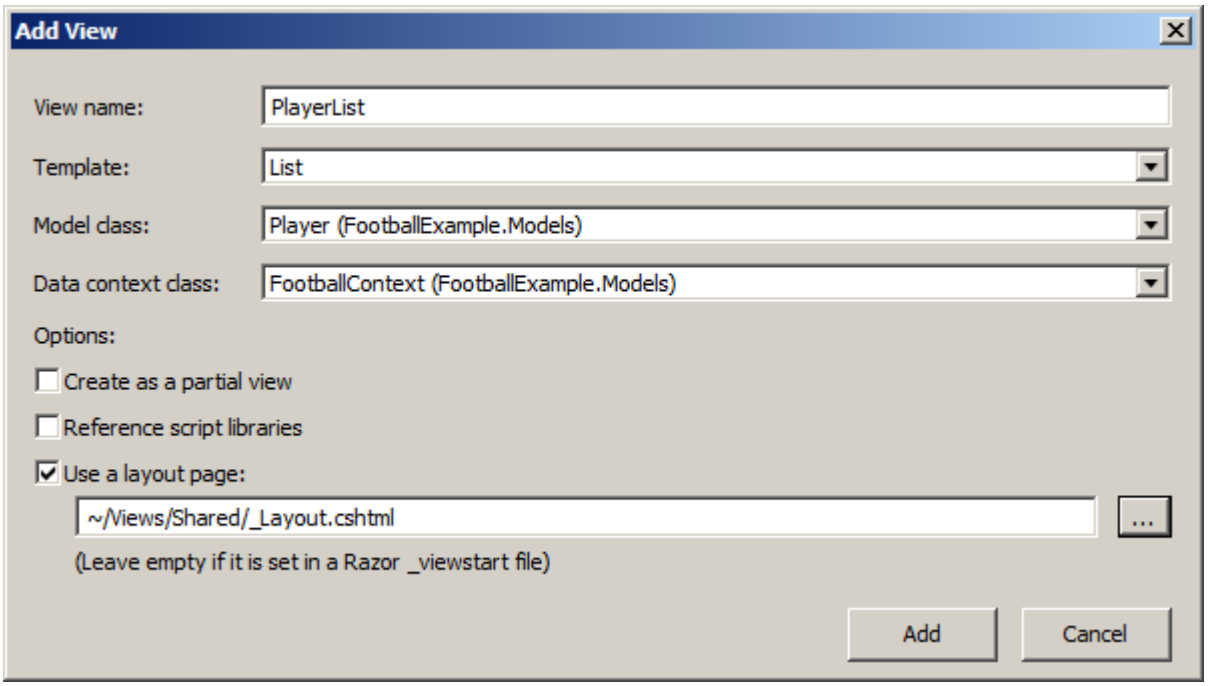
```
private readonly FootballContext _db = new FootballContext();
public ActionResult Index()
{
    return View(_db.Players);
}
```

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:

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

<div>
    <h3>Список игроков</h3>
    <table>
        <tr>
            <td><p>Имя</p></td>
            <td><p>Возраст</p></td>
            <td><p>Позиция</p></td>
            <td></td>
        </tr>
        @foreach (var p in Model)
        {
            <tr>
                <td><p>@p.Name</p></td>
                <td><p>@p.Age</p></td>
                <td><p>@p.Position</p></td>
                <td><p><a href="/Home/Vote/@p.Id">Голосовать</a></p></td>
            </tr>
        }
    </table>
</div>
```

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

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.

10. 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. Изменим её 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("Главная", "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.

```
<!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()
    <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";
}
<!-- здесь остальное содержание -->
@section Footer
{
    Все права защищены. Syte Corp. 2012.
}
```

Но при таком подходе, если у нас есть куча представлений, и мы вдруг захотели определить новую секцию на мастер-странице, нам придётся 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>
```

10.1. 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.

11. Работа с формами

Фреймворк MVC предоставляет большой набор встроенных html-хелперов, 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.

Html.BeginForm

Создает обычную html-форму, которая по нажатию на кнопку отправляет все введенные данные запросом POST на указанный адрес.

```
@using (Html.BeginForm("Vote", "Home", FormMethod.Post))
{
    <input type="hidden" value="@ViewBag.BookId" name="BookId" />
```

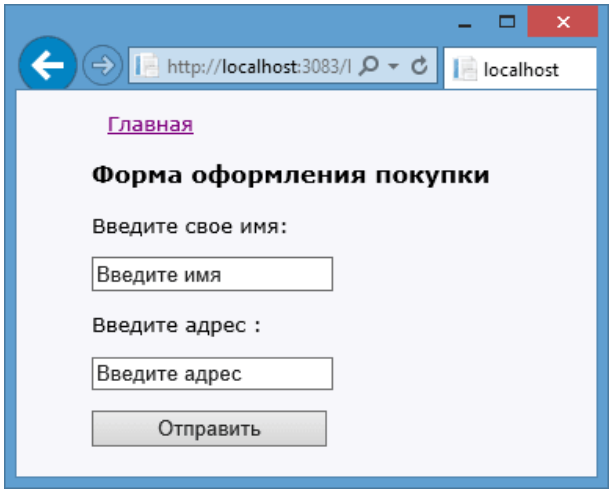
```
    <table>
        <tr>
            <td><p>Введите свое имя </p></td>
            <td><input type="text" name="Person" /> </td>
        </tr>
        <tr>
            <td><p>Введите адрес :</p></td>
            <td><input type="text" name="Address" /> </td>
        </tr>
        <tr>
            <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>.

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>
}
```



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>
```

Другие версии хелпера TextArea позволяют указать число строк и столбцов, определяющих размер текстового поля.

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

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" />
```

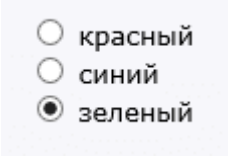
Html.RadioButton

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 объектовSelectListItem, 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 хелперSelectList. 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>
```

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 однйо 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 одинаковое значение атрибута name, но разное для атрибута value. А метод, принимающий форму, может выглядеть следующим образом:

```
[HttpPost]
public ActionResult MyAction(string product, string action)
```

```
{
    if (action == "add")
    {

    }
    else if (action == "delete")
    {

    }
    // остальной код метода
}
```

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

12. 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.

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

<h2>Изменение игрока</h2>

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

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

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

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

        <p>
            Позиция<br />
            @Html.DropDownListFor(model => model.Position, new SelectList(ViewBag.Posi-
tions)))
        </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("К списку игроков", "Index")
</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 FootballExample.Models.Player. То есть, в вызове @Html.TextBoxFor(m=>m.Name) параметр m представляет переменную модели Player. А лямбда-выражение m=>m.Name указывает, что данный хелпер будет генерировать текстовое поле для свойства Name. Таким образом, хелпер @Html.TextBoxFor(m=>m.Name) сгенерирует текстовое поле <input id="Name" name="Name" type="text" value="" />

Для каждого базового встроенного хелпера имеется свой строго типизированный хелпер:

- Html.CheckBoxFor
- Html.HiddenFor
- Html.LabelFor
- Html.PasswordFor
- Html.RadioButtonFor
- Html.TextBoxFor
- Html.TextAreaFor

12.1. Шаблонные хелперы

Кроме базовых html-хелперов, рассмотренных в прошлом разделе генерирующих определенные элементы разметки html, фреймворк ASP.NET MVC также имеет шаблонные (или шаблонизированные) хелперы. В отличие от рассмотренных 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()

13. Модели со сложной структурой

До сих пор мы работали только с объектами типа Player, при этом навигационное поле Team, указывающее на принадлежность игрока к определенной команде мы намеренно пропускали. Теперь постараемся восполнить этот пробел.

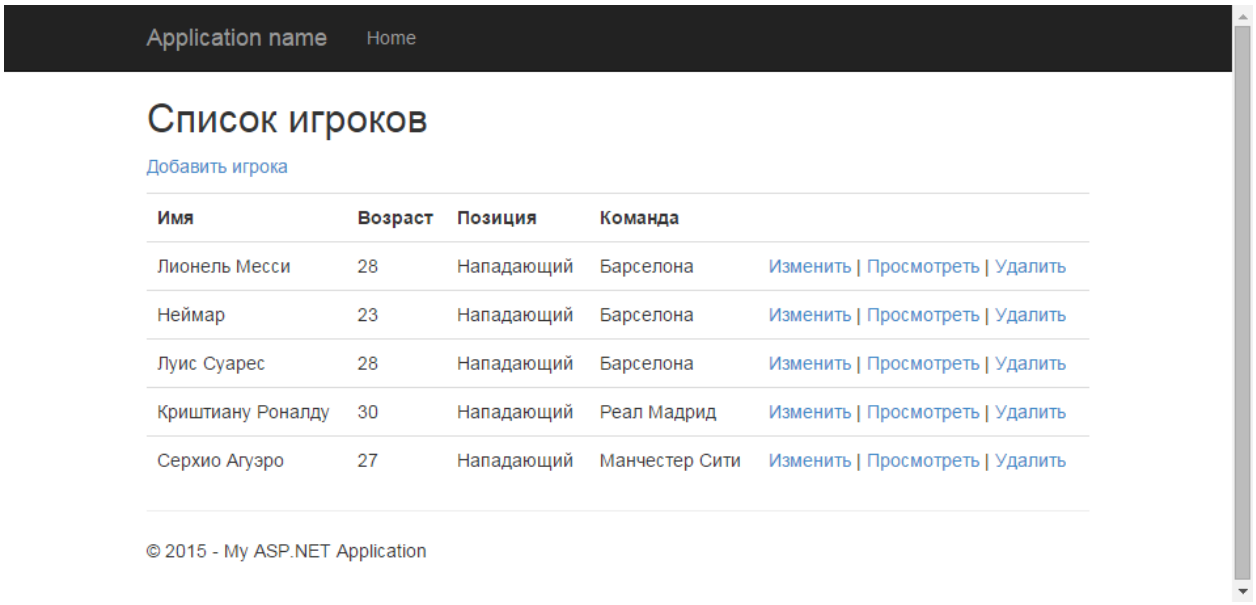
Изменим наш контроллер и определим в нем вывод всех игроков на страницу:

```
public class HomeController : Controller
{
    // создаем контекст данных
    private readonly FootballContext _db = new FootballContext();
    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<FootballExample.Models.Player>

@{
    ViewBag.Title = "Список игроков";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<h2>Index</h2>

<p>
    @Html.ActionLink("Добавить игрока", "Create")
</p>
<table class="table">
    <tr>
        <th>Имя</th>
        <th>Возраст</th>
        <th>Позиция</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("Изменить", "Edit", new { id=item.Id }) |
            @Html.ActionLink("Просмотреть", "Details", new { id=item.Id }) |
            @Html.ActionLink("Удалить", "Delete", new { id=item.Id })
        </td>
    </tr>
}
</table>
```



Для реализации операции просмотра сведений о каждом игроке реализуем следующий метод в контроллере:

```
public ActionResult Details(int? id)
{
    var player = _db.Players.Include(p => p.Team).FirstOrDefault(p => p.Id == id);

    if (player == null)
    {
        return HttpNotFound();
    }

    return View(player);
}
```

И создадим представление для вывода информации на экран

```
@model FootballExample.Models.Player
```

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

<h2>Сведения об игроке</h2>

<div>
    <h4>@Html.DisplayFor(model => model.Name)</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>Команда:</dt>
        <dd>@Html.DisplayFor(model => model.Team.Name)</dd>
        <dt>Возраст:</dt>
        <dd>@Html.DisplayFor(model => model.Age)</dd>
        <dt>Позиция:</dt>
        <dd>@Html.DisplayFor(model => model.Position)</dd>
    </dl>
</div>
<p>
    @Html.ActionLink("Edit", "Edit", new { id = Model.Id }) |
    @Html.ActionLink("Back to List", "Index")
</p>
```

Подобным образом можно вывести список команд. Создадим новый контроллер TeamsController.

```
    public class TeamsController : Controller
    {
        private readonly FootballContext _db = new FootballContext();

        public ActionResult Index()
        {
            return View(_db.Teams.ToList());
        }
    }
```

Представление схоже с представлением со списком игроков:

```
@model IEnumerable<FootballExample.Models.Team>

@{
    ViewBag.Title = "Список команд";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<h2>Список команд</h2>

<p>
    @Html.ActionLink("Добваить команду", "Create")
</p>
<table class="table">
    <tr>
        <th>Название</th>
        <th>Тренер</th>
        <th></th>
    </tr>

@foreach (var item in Model) {
    <tr>
```

```
        <td>
            @Html.DisplayFor(modelItem => item.Name)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Coach)
        </td>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.Id }) |
            @Html.ActionLink("Details", "Details", new { id=item.Id }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.Id })
        </td>
    </tr>
}

</table>
```

У нас в модели Team свойство Players, которое призвано хранить связанных с командой игроков. Используем его. Например, выведем все данные о команде используя метод Details, в том числе о ее игроках.

```
        public ActionResult Details(int? id)
        {
            if (id == null)
            {
                return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
            }
            var team = _db.Teams.Include(t => t.Players).FirstOrDefault(t => t.Id == id);
            if (team == null)
            {
                return HttpNotFound();
            }
            return View(team);
        }
```

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

```
@model FootballExample.Models.Team

@{
    ViewBag.Title = "Сведения об команде";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<h2>Сведения об команде</h2>

<div>
    <h4>@Model.Name</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>Тренер:</dt>
        <dd>
            @Html.DisplayFor(model => model.Coach)
        </dd>
        <dt>Игроки</dt>
        <dd>
            <ul>
                @foreach (var player in Model.Players)
                {
                    <li>@player.Name (@player.Position)</li>
                }
            </ul>
        </dd>
```

```
    </dl>
</div>
<p>
    @Html.ActionLink("Edit", "Edit", new { id = Model.Id }) |
    @Html.ActionLink("Back to List", "Index")
</p>
```

13.1. 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 Edit(int? id)
{
    if (id == null)
    {
        return HttpNotFound();
    }

    var player = _db.Players.Find(id);
    if (player != null)
    {
        var teams = new SelectList(_db.Teams, "Id", "Name", player.TeamId);
        ViewBag.Teams = teams;
        return View(player);
    }

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

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.

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

```
@model FootballExample.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<br />
            @Html.EditorFor(model => model.Name)
        </p>

        <p>
            Возраст <br />
```

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

        <p>
            Позиция<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>
```

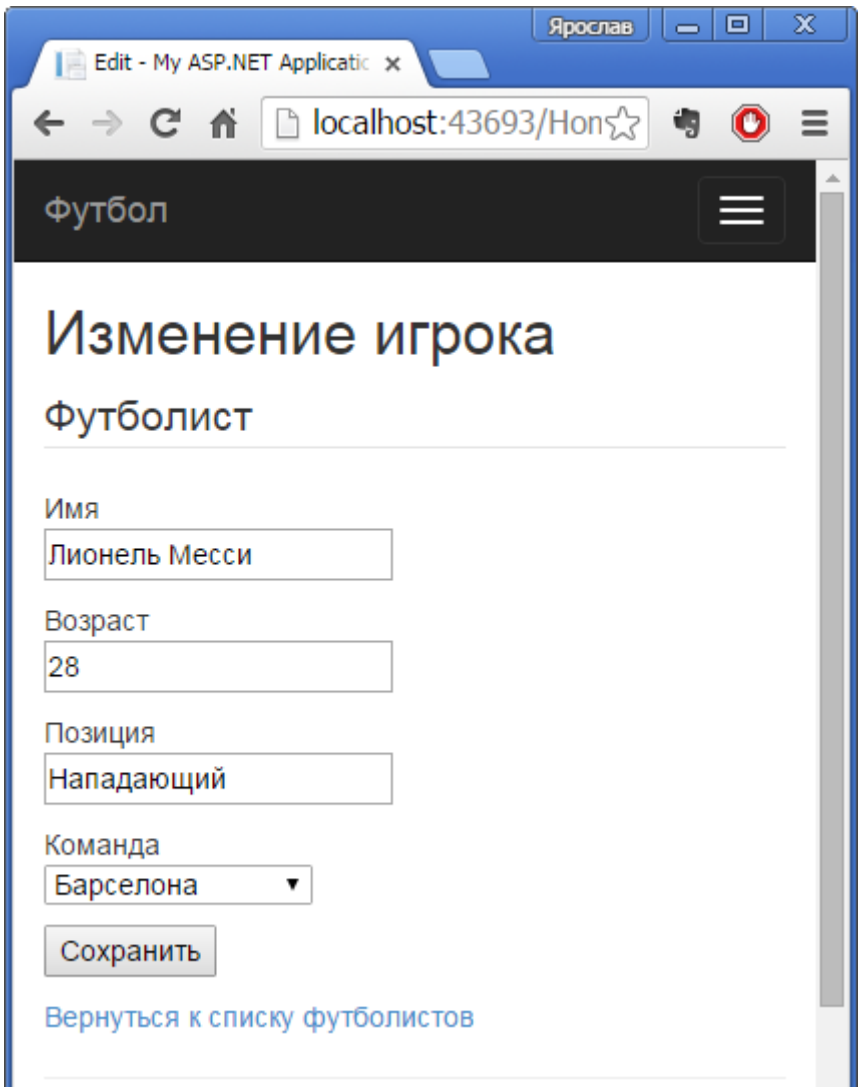
Так как уникальный идентификатор id игрока нам не надо редактировать, то поле для его отображения сделаем скрытым, то есть воспользуемся хелпером Html.HiddenFor.

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

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

С помощью строки db.Entry(player).State = EntityState.Modified; мы указываем, что объект существует player 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(player.State = EntityState.Modified;, то мы тем самым указываем методу db.SaveChanges(), что надо сгенерировать и выполнить команду UPDATE для обновления модели в БД.



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

13.2. Добавление и удаление модели

Добавление модели

В предыдущей теме мы посмотрели, как редактировать модель. Продолжим работу с моделью Player и теперь посмотрим, как мы можем ее добавлять и удалить из БД. 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()
{
    var teams = new SelectList(_db.Teams, "Id", "Name");
    ViewBag.Teams = teams;

    return View();
}

[HttpPost]
public ActionResult Create(Player player)
{
```

```
            _db.Players.Add(player);
            _db.SaveChanges();

            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 FootballExample.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>
```

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

При получении модели player в действии Create метод db.Players.Add(player) 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(Player player)
```

```
{
    _db.Entry(player).State = EntityState.Added;
    _db.SaveChanges();

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

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)
{
    Player p = _db.Players.Find(id);
    if (p != null)
    {
        _db.Players.Remove(p);
        _db.SaveChanges();
    }
    return RedirectToAction("Index");
}
```

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

```
public ActionResult Delete(int id)
{
    Player p = new Player { Id = id };
    _db.Entry(p).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 Player p = db.Players.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)
{
    var book = _db.Players.Find(id);
    if (book == null)
    {
        return HttpNotFound();
    }

    return View(book);
}
```

```
        [HttpPost, ActionName("Delete")]
        public ActionResult DeleteConfirm(int? id)
        {
            var player = _db.Players.Find(id);
            _db.Players.Remove(player);
            _db.SaveChanges();

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

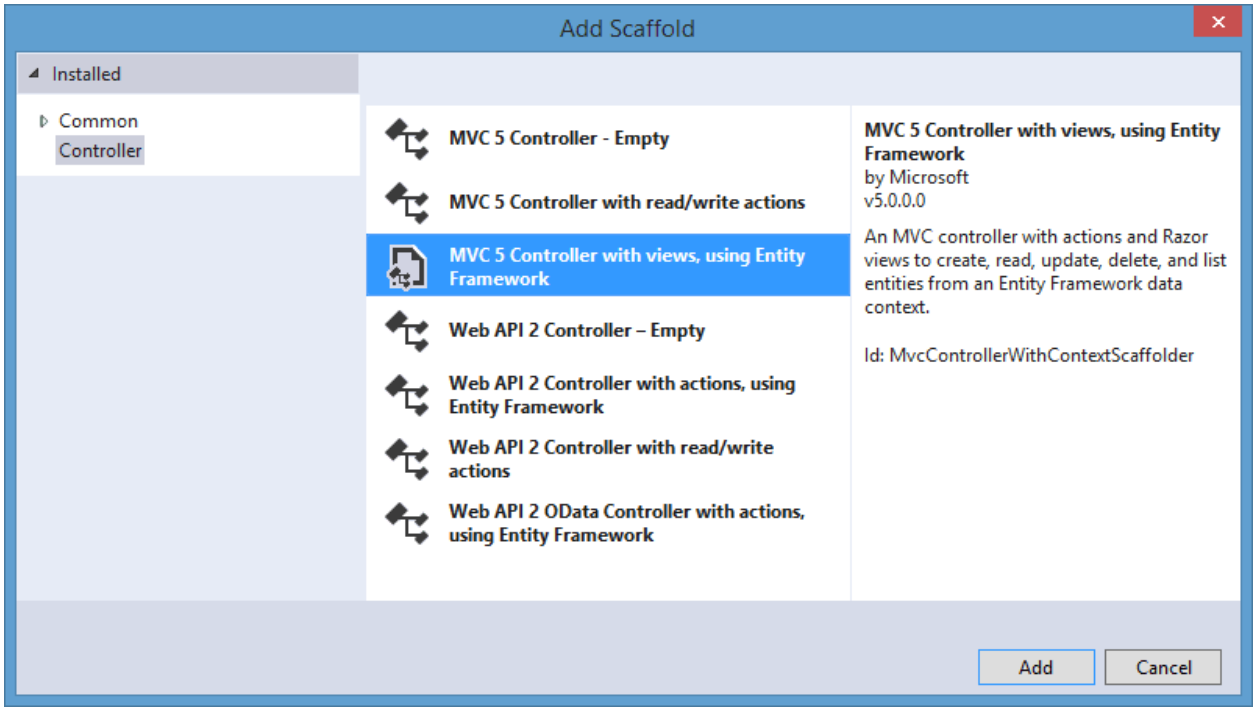
Теперь вместо одного метода Delete целых два. Атрибут ActionName("Delete") указывает, что метод DeleteConfirm будет восприниматься как действие Delete. Первый метод передает удаляемую модель в представление. На представлении с помощью нажатия кнопки мы сможем подтвердить удаление. И удаляемый id уйдет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

```
@model FootballExample.Models.Player

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

<h2>Удаление игрока</h2>

<h3>Вы действительно хотите удалить этого игрока</h3>
<div>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Name)
        </dt>

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

    @using (Html.BeginForm())
    {
        @Html.AntiForgeryToken()

        <div class="form-actions no-color">
            <input type="submit" value="Delete" class="btn btn-default" /> |
            @Html.ActionLink("К списку игроков", "Index")
        </div>
    }
</div>
```

14. 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.

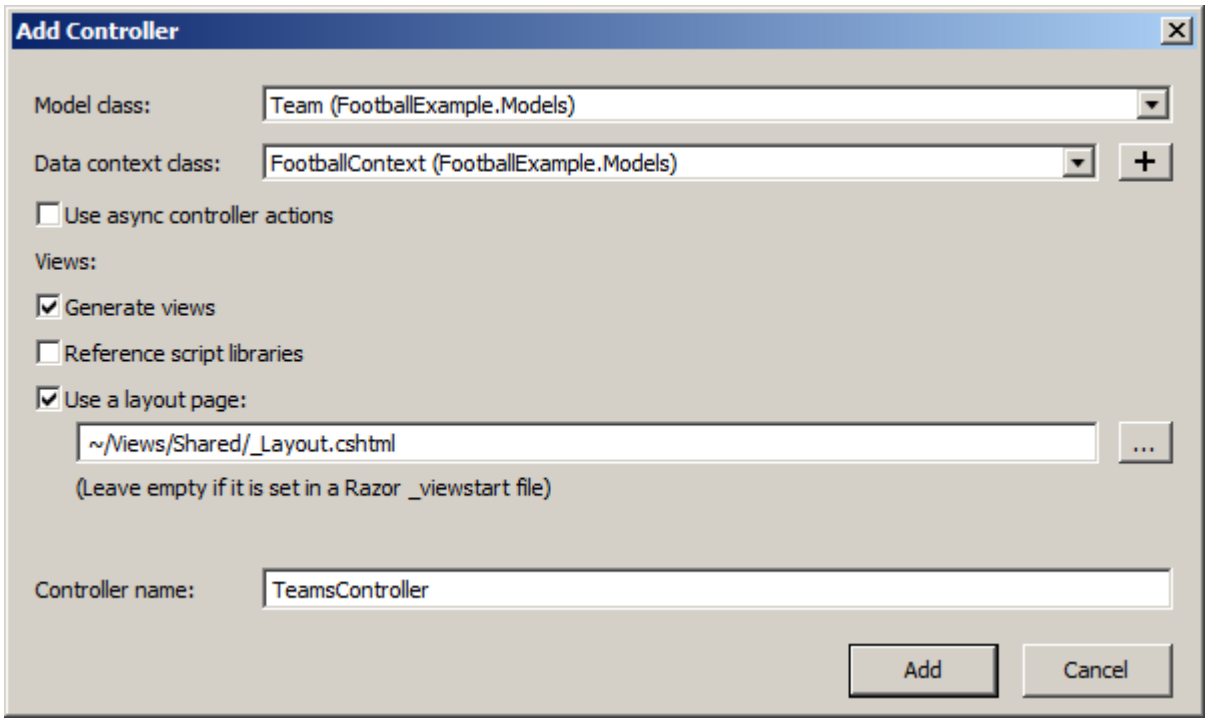
- Model class: класс модели. Выберем созданную ранее модель Book (либо какую-то другую имеющуюся модель)

- 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.

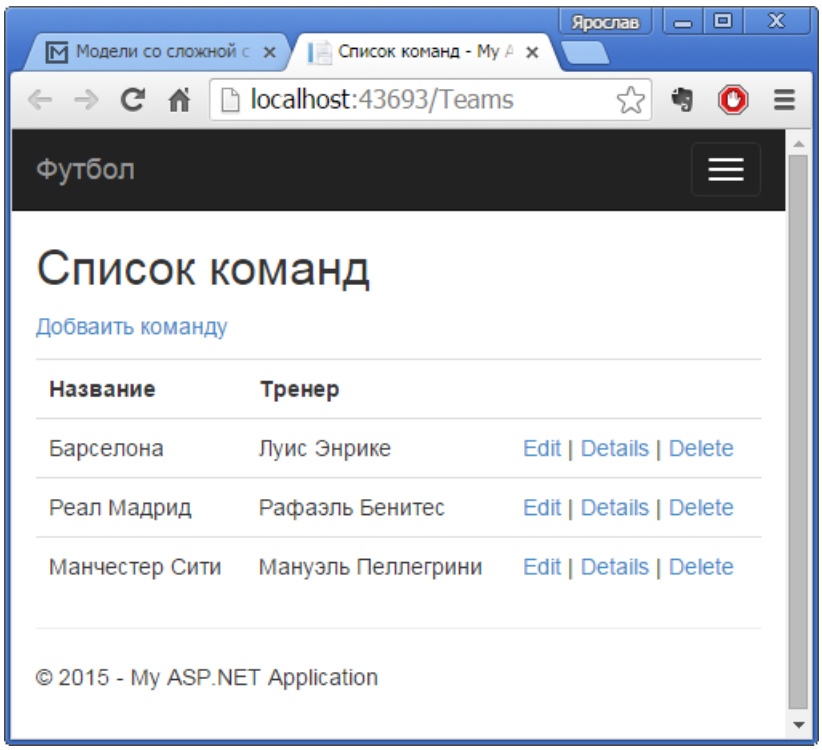
- Reference script libraries: будут ли подключать представления библиотеки jquery и другие необходимые файлы javascript

- Use a layout page: будут ли генерируемые представления использовать мастер-страницу



Установив все опции, нажмем кнопку Add, и в проект будет добавлен новый контроллер.

А в папке Views/Teams 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:



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.

15. Создание пагинации

Одной из распространенных задач при работе с данными является проблема пагинации или разделения данных на несколько блоков - страниц, которые выводятся в 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.

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<Player> Players { get; set; }
    public PageInfo PageInfo { get; set; }
}
```

Так как вместе с данными игроков потребуется также использовать в представлении модель пагинации, то они инкапсулируются классом IndexViewModel.

Теперь определим контроллер и его метод для вывода данных:

```
public ActionResult Index(int page = 1)
{
    var players = _db.Players.Include(p => p.Team).ToList();
    var pageSize = 3; // количество объектов на страницу
    var playersPerPages = players.Skip((page - 1) * pageSize).Take(pageSize);
    var pageInfo = new PageInfo { PageNumber = page, PageSize = pageSize, TotalItems =
players.Count() };
    var ivm = new IndexViewModel { PageInfo = pageInfo, Players = playersPerPages };

    return View(ivm);
}
```

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:

```
namespace FootballExample.Helpers
{
    public static class PagingHelpers
    {
        public static MvcHtmlString PageLinks(this HtmlHelper html,
            PageInfo pageInfo, Func<int, string> pageUrl)
        {
            var result = new StringBuilder();
            for (var i = 1; i <= pageInfo.TotalPages; i++)
            {
                var 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);
            }
            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.

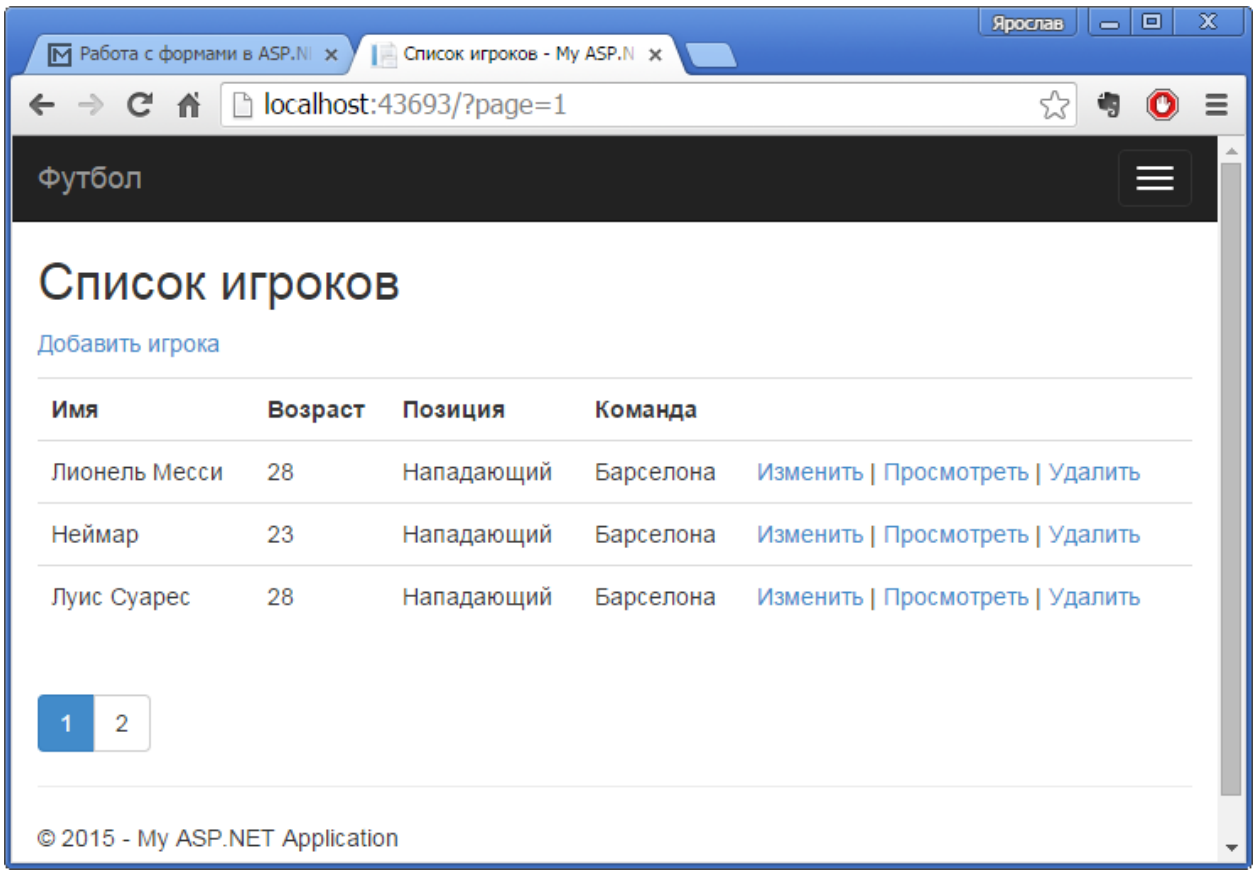
И теперь определим представление, которое использовать пагинацию:

```
@model FootballExample.Models.IndexViewModel
@using FootballExample.Helpers


@{
    ViewBag.Title = "Список игроков";
```

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

<h2>Список игроков</h2>

<p>
    @Html.ActionLink("Добавить игрока", "Create")
</p>
<table class="table">
    <tr>
        <th>Имя</th>
        <th>Возраст</th>
        <th>Позиция</th>
        <th>Команда</th>
        <th></th>
    </tr>

    @foreach (var item in Model.Players)
    {
        <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("Изменить", "Edit", new {id = item.Id}) |
                @Html.ActionLink("Просмотреть", "Details", new {id = item.Id}) |
                @Html.ActionLink("Удалить", "Delete", new {id = item.Id})
            </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 FootballExample.Helpers И хелпер пагинации Html.PageLinks сделают всю работу по формированию блока ссылок в соответствии с заложенной в нем логикой.

В итоге получим примерно следующую картину:



16. Фильтрация данных

Рассмотрим, как фильтровать данные по определенным критериям в ASP.NET MVC.

Допустим, при выводе списка футболистов нам надо также предусмотреть возможность для их фильтрации по некоторым критериям, например, по команде и по позиции на поле.

Так как мы будем выводить в одном представлении и список игроков, и критерии для выбора, которые представляют списки команд и позиций, то добавим в проект специальную модель:

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

namespace NavigationProperty.Models
{
    public class PlayersListViewModel
    {
        public IEnumerable<Player> Players { get; set; }
        public SelectList Teams { get; set; }
        public SelectList Positions { get; set; }
    }
}
```

Три свойства - для списка игроков, команд и позиций. К классу также можно добавить свойства, относящиеся к пагинации списка игроков.

Для фильтрации данный данных модели добавим в представление форму, которая с помощью запросов GET посылает выбранные в выпадающих списках значения на сервер.

```
@using (Html.BeginForm())
{
<div class="form-inline">
    <label class="control-label">Команда: </label>
    @Html.DropDownList("team", Model.Teams, new { @class = "form-control" })

    <label class="control-label">Позиция игрока: </label>
    @Html.DropDownList("position", Model.Positions, new { @class = "form-control" })

    <input type="submit" value="Фильтр" class="btn btn-default" />
</div>
}
```

Теперь на стороне контроллера определим метод, который будет выводить список игроков и производить фильтрацию:

```
private readonly List<string> _positions = new List<string>
        {
            "", "Нападающий", "Полузащитник", "Защитник", "Вратарь"
        };
        public ActionResult Index(int? team, string position, int page = 1)
        {
            var players = _db.Players.Include(p => p.Team);
            if (team != null && team != 0)
            {
                players = players.Where(p => p.TeamId == team);
            }

            if (!String.IsNullOrEmpty(position) && !position.Equals("Все"))
            {
                players = players.Where(p => p.Position == position);
            }

            var teams = _db.Teams.ToList();
            // устанавливаем начальный элемент, который позволит выбрать всех
            teams.Insert(0, new Team { Name = "Все", Id = 0 });

            var pageSize = 3; // количество объектов на страницу
            var playersPerPages = players.ToList().Skip((page - 1) *
pageSize).Take(pageSize);
            var pageInfo = new PageInfo { PageNumber = page, PageSize = pageSize, To-
talItems = players.Count() };

            var plvm = new PlayersListViewModel
            {
                Players = playersPerPages.ToList(),
                Teams = new SelectList(teams, "Id", "Name"),
                Positions = new SelectList(_positions),
                PageInfo = pageInfo
            };

            return View(plvm);        }
```

При обращении к методу Index вне зависимости были ли переданы параметры team и position, все равно в представление будет выводиться список игроков. Если определенный параметр был передан, то к объекту IQueryable, представляющего выборку, добавляется выражение Where: players = players.Where(p=>p.TeamId==team);.

Для формирования выпадающего списка команд в представлении, получаем его из БД:

```
List<Team> teams = db.Teams.ToList();
```

Для возможности выбора игроков из любой команды, добавляем в этот список новый пункт:

```
teams.Insert(0, new Team { Name = "Все", Id = 0 });
```

Все полученные и созданные списки используются для формирования объекта PlayersListViewModel, который передается в представление.