ASP.NET MVC представляет собой платформу для создания сайтов и веб-приложений с использованием паттерна (или шаблона) MVC (model - view - controller).

Работа над новой платформой была начата в 2007 году, а в 2009 году появилась первая версия. В итоге к текущему моменту (2012 год) уже было выпущено 4 версии платформы, а сам фреймворк обрел большую популярность по всему миру благодаря своей гибкости и адаптивности.

Шаблон MVC, лежащий в основе новой платформы, подразумевает взаимодействие трех компонентов: контроллера (controller), модели (model) и представления (view). Что же представляют эти компоненты?

**Контроллер** (controller) представляет класс, с которого собственно и начинается работа приложения. Этот класс обеспечивает связь между моделью и представлением. Получая вводимые пользователем данные, контроллер исходя из внутренней логики при необходимости обращается к модели и генерирует соответствующее представление.

**Представление** (view) - это собственно визуальная часть или пользовательский интерфейс приложения - например, html-страница, через которую пользователь, зашедший на сайт, взаимодействует с веб-приложением.

**Модель** (model) представляет набор классов, описывающих логику используемых данных.

Общую схему взаимодействия упрощенно можно представить следующим образом:

**ASP.NET MVC и ASP.NET Web Forms**

ASP.NET MVC является в некотором роде конкурентом для традиционных веб-форм и имеет по сравнению с ними следующие преимущества:

* **Разделение ответственности**. В MVC приложение состоит из трех частей: контроллера, представления и модели, каждая из которых выполняет свои специфичные функции. В итоге приложение будет легче поддерживать модифицировать в будущем.
* В силу разделения ответственности приложения mvc обладают **лучшей тестируемостью**. И мы можем тестировать отдельные компоненты независимо друг от друга.
* **Соответствие протоколу HTTP**. Приложения MVC в отличие от веб-форм не поддерживают объекты состояния (ViewState). Ясность и простота платформы позволяют добиться большего контроля над работой приложения
* **Гибкость**. Вы можете настраивать различные компоненты платформы по своему усмотрению. Изменять какие-либо части конвейера работы MVC или адаптировать его к своим нуждам и потребностям.

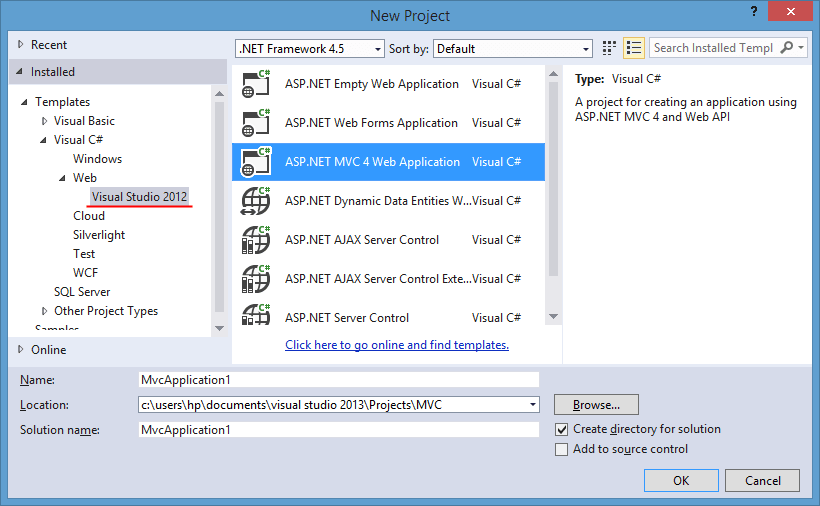
В то же время не стоит однозначно сбрасывать со счетов ASP.NET WebForms. Поскольку она также имеет свои сильные стороны, например, модель событий, которая будет ближе тем разработчикам, которые ранее занимались созданием клиентских приложений.

В традиционных веб-формах вы имеете контроль над разметкой и можете в реальном времени в визуальном редакторе Visual Studio увидеть, как будет выглядеть та или иная страница. При работе с MVC Visual Studio подобного не позволяет делать.

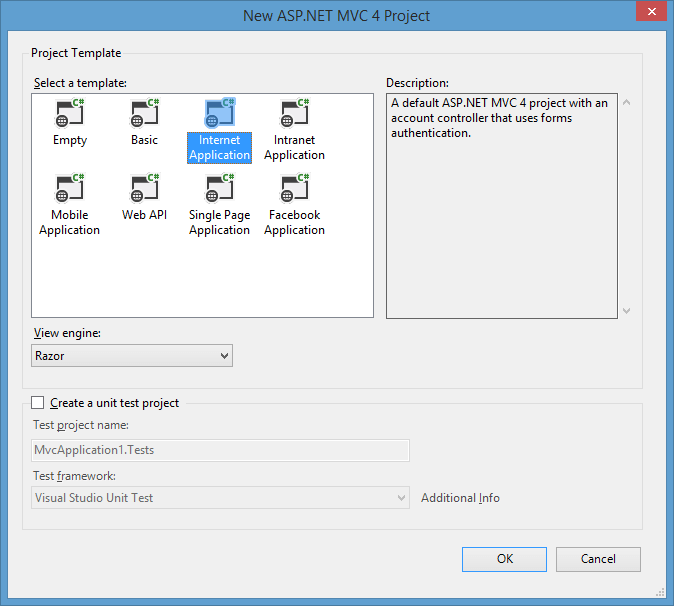
В любом случае вы вольны выбирать ту платформу, которая приходится вам больше по душе. И если у вас написаны объемные проекты с применением традиционных веб-форм, возможно, стоит продолжать с ними работать. Тем более, что ASP.NET Web Forms еще не умирает и также продолжает развиваться.

Итак, чтобы создавать приложения на ASP.NET MVC нам потребуется среда разработки - Microsoft Visual Studio Community 2013 или Microsoft Visual Studio Express 2013 for Web, которые можно найти на странице [Visual Studio 2013](https://www.visualstudio.com/ru-ru/downloads/download-visual-studio-vs#DownloadFamilies_2). Также мы можем использовать один из выпусков Visual Studio 2012, которые тоже включают средства разработки для MVC 4.

После установки всего необходимого инструментария мы можем создать первое приложение. Итак, откроем Microsoft Visual Studio Community 2013 (или другой имеющийся выпуск Visual Studio 2012 / 2013). В меню **File (Файл)** выберем пункт **New Project... (Создать проект)**. Перед нами откроется диалоговое окно создания проекта. Cреди типов проекта выбрем Web->Visual Studio 2012, а в центральном поле - шаблон ASP.NET MVC 4 Web Application:



После выбора типа проекта перед нами откроется окно выбора шаблона нового приложения:



Выберем шаблон **Internet Application**. Что представляют все эти шаблоны?

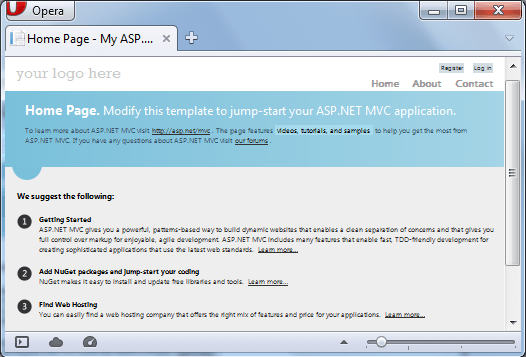
#### Шаблоны приложения MVC 4

* **Internet Application**. Данный шаблон представляет некоторую базовую функциональность, например, управление учетными записями, которую будет содержать новое приложение.
* **Intranet Application**. Данный шаблон похож на шаблон Internet Application. Отличие состоит в том, что в Intranet Application для управления учетными записями используется аутентификация Windows, а не система ASP.NET Membership.
* **Empty**. Этот шаблон представляет абсолютно чистое приложение без каких-либо файлов и структуры папок.
* **Basic**. Этот шаблон также представляет пустой проект. Правда, он содержит базовые каталоги, CSS, инфраструктуру приложения MVC. Ранее в MVC 3 этот шаблон назывался Empty.
* **Mobile Application**. Это новинка MVC 4, предназначенная для создания веб-приложений для мобильных платформ.
* **Web API**. Предназначен для создания сервисов и для работы с фреймворком Web API.
* **Single Page Application**. Шаблон, предназначенный для создания сайтов-одностраничников.
* **Facebook Application**. Предназначен для создания веб-приложений под социальную сеть Facebook.

Кроме того, данное диалоговое окно предлагает нам выбрать движок представления в секции **View Engine** и задать опции тестирования.

При выборе движка представления нам, как и в MVC 3, доступны две опции - движок **Razor** и ASPX (традиционный движок Web Forms). В данном случае мы выберем Razor. Опции тестирования мы пока трогать не будем.

Итак, нажимаем кнопку OK и создаем новый проект. Мы сразу же можем запустить проект на выполнение и нам предстанет чудная картинка в браузере с нашим приложением:



Таким образом, весь функционал, который содержит проект по умолчанию, реализуется в такое приложение. Теперь давайте посмотрим, что представляет структура проекта.

### Структура проекта MVC 4

Справа в окне Solution Explorer (Обозреватель решений) мы можем увидеть структуру проекта MVC 4. Тот, кто раньше работал с предыдущими версиями MVC, заметит некоторые отличия. Итак, пройдемся по папкам и файлам проекта.

#### App\_Data

В этой папке хранятся все необходимые файлы и ресурсы, например, базы данных, используемые приложением. После развертывания приложения только непосредственно приложение может работать с этой папкой, доступ же простых пользователей в эту папку запрещен.

#### Файл Global.asax и папка App\_Start

В mvc 4 была добавлена папка App\_Start. Она включает весь функционал конфигурации приложения, который в предыдущих версиях содержался в файле Global.asax, а теперь перенесен в набор статичных классов, вызываемых в Global.asax. Эти статичные классы содержат некоторую логику инициализации приложения, выполняющуюся при запуске.

#### Файл Web.config

Файл конфигурации приложения, который находится в корневой папке приложения

#### Content

Содержит некоторые вспомогательные файлы, которые не включают код на c# или javascript, и которые развертываются вместе с приложением. В частности, здесь могут размещаться файлы стилей css. Так, в этой папке вы увидите файл Site.css, который содержит стили приложения, а также папку с темами, включающую стили css и изображения для определенных тем.

#### Controllers

Содержит контроллеры - классы, отвечающие за работу приложения. По умолчанию здесь находятся два контроллера - HomeController и AccountController.

#### Папки Images и Scripts

Папки Images и Scripts содержат соответственно изображения и скрипты на JavaScript, используемые в приложении. По умолчанию эти папки уже содержат файлы, в частности, в папку Scripts уже помещены файлы библиотеки jQuery.

#### Models

Содержит модели, используемые приложением. По умолчанию здесь определена одна модель - AccountModel, которая представляет отдельную учетную запись.

#### Views

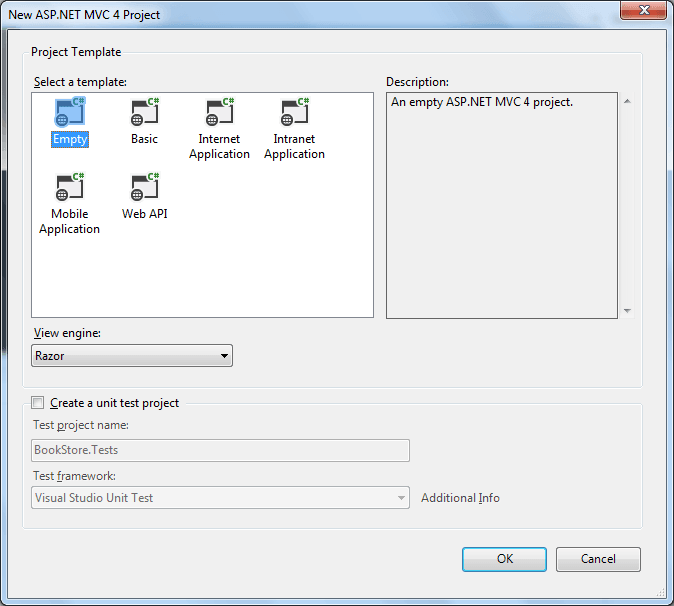
Здесь размещаются представления. Представления группированы по папкам, каждая из которых соответствует одному контроллеру. После получения и обработки запроса контроллер, отправляет одно из этих представлений, заполненных некоторыми данными, клиенту. Кроме того, имеется папка общих для контроллеров представлений - папка Shared

Итак, мы посмотрели на некоторые базовые части проекта и теперь создадим первое приложение.

Сейчас мы создадим первое примитивное приложение на mvc 4, которое призвано дать некоторое начальное понимание работы mvc. В качестве шаблона проекта мы выберем шаблон Empty и создадим мини-приложение с нуля.

Что будет делать наше приложение? Приложение будет представлять примитивную эмуляцию книжного магазина, в котором мы сможем просматривать список книг и оформлять покупку.

Итак, откроем Visual Studio или Visual Web Developer, в меню выберем **File -> New Project..** и среди шаблонов проектов выберем шаблон**ASP.NET MVC 4 Web Application** и как-нибудь назовем новый проект, например, BookStore. Затем в окне создания нового проекта ASP.NET MVC 4 Web Application выберем шаблон Empty:



После этого будет создан по сути пустой проект, в котором еще ничего нет. Нам предстоит наполнить его.

Во-первых, нам предстоит работать с данными, которые должны быть представлены некими моделями. Для нашей задачи мы можем выделить две области данных: информация о книге и информация, связанная с оформлением покупки. Соответственно, мы создадим две модели. Итак, в нашем приложении уже определена папка **Models**. В ней будут находиться наши модели. Нажмем на эту папку правой кнопкой мыши и в появившемся меню выберем **Add->Class...**. Назовем новый класс или модель **Book** и добавим в него код, описывающий модель книги:

|  |  |
| --- | --- |
|  | namespace BookStore.Models  {      public class Book      {          // ID книги          public int Id { get; set; }          // название книги          public string Name { get; set; }          // автор книги          public string Author { get; set; }          // цена          public int Price { get; set; }      }  } |

Теперь таким же образом создадим класс для оформления покупки - класс **Purchase**:

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

**Условности при создании моделей**

В данном случае надо сказать, что в данном случае действуют некоторые условности. Поскольку база данных и соответствующая ей модель должна иметь первичный ключ (Primary Key), то мы его должны естественно определить. У нас есть два пути определения ключа. В данном случае мы использовали определение ключа по названию свойства. Для этого модель должна содержать свойство с именем**Имя\_моделиId** или просто **Id**. Так, у нас в модели Purchase определено свойство PurchaseId, то есть данное свойство является первичным ключом. Второй способ состоял в определении ключа с помощью атрибута Key, установленным над нужным свойством. Более подробно об этом мы поговорим в главе, где будем обсуждать модели.

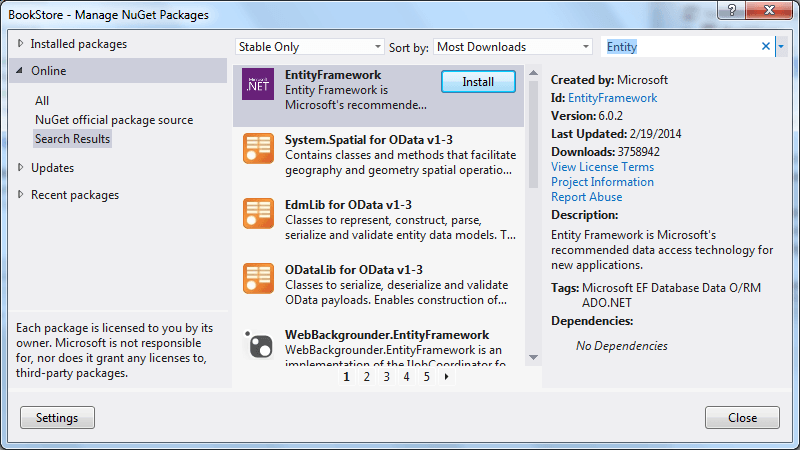
**Создание контекста данных и EntityFramework**

Для более удобного доступа к данным рекомендуется использовать фреймворк Entity Framework, хотя его использование необязательно и всецело зависит от предпочтений разработчика. Этот фреймворк позволяет абстрагироваться от структуры конкретной базы данных и вести все операции с данными через модель.

Поскольку мы создали проект по шаблону Empty, который практически не содержит никаких файлов, то нам надо будет добавить в проект файлы EntityFramework для взаимодействия с БД. Если бы мы создавали проект по другому шаблону, например, Basic или Internet Application, то библиотека EntityFramework уже была бы добавлена по умолчанию.

Итак, добавим в проект поддержку EntityFramework и для этого нажмем правой кнопкой мыши в структуре проекта на узел References (который содержит подключенные в проект библиотеки) и в появившемся меню выберем **Manage NuGet Packages...**

В окне управления пакетами NuGet в правом верхнем углу введите в поле поиска **Entity** и нажмите Enter. После этого все найденные пакеты, которые имеют отношение к запросу, будут отображены в среднем столбце, как показано на рисунке:



Затем устанавливаем пакет, и после того, как закончится установка, мы можем использовать все преимущества EntityFramework для управлением данными.

После установки создадим контекст данных. Он нужен для облегчения доступа к БД на основе некоторой модели. Добавим в папку Models новый класс **BookContext** со следующим содержанием:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | using System;  using System.Collections.Generic;  using System.Web;  using System.Data.Entity;    namespace BookStore.Models  {      public class BookContext : DbContext      {          public DbSet<Book> Books { get; set; }          public DbSet<Purchase> Purchases { get; set; }      }  } |

Чтобы создать контекст, нам надо унаследовать новый класс от класса **DbContext**. Свойства типа public DbSet<Book> Books { get; set; }помогают получать набор данных определенного типа (в данном случае типа Book).

В итоге EntityFramework сам возьмет на себя заботу по созданию базы данных. То есть в данном случае мы будем использовать так называемый подход **Code First** - у нас есть модели, и по ним фреймворк будет создавать таблицы в базе данных.

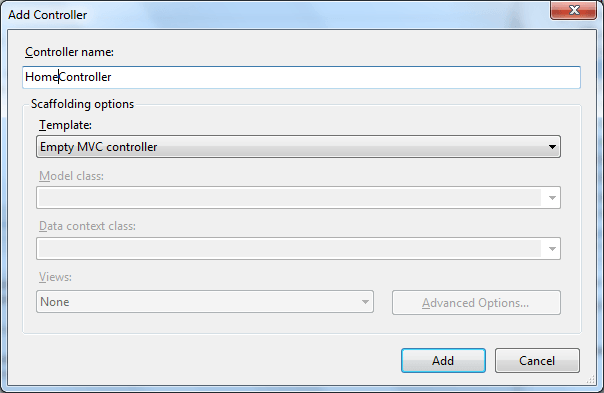
Наконец, укажем строку подключения. Откроем файл *web.config*, найдем секцию *configSections* и после нее вставим секцию*connectionStrings*:

|  |  |
| --- | --- |
| 1  2  3  4 | <connectionStrings>      <add name="BookContext" connectionString="Data Source=(LocalDB)\v11.0;AttachDbFilename='|DataDirectory|\Bookstore.mdf';Integrated Security=True"   providerName="System.Data.SqlClient" />  </connectionStrings> |

Тем самым мы определяем путь к базе данных, которая потом будет создаваться. Выражение |DataDirectory| представляет заместитель, который указывает, что база данных будет создаваться в проекте в папке App\_Data. Позднее мы подробнее разберем настройки подключения к БД.

## Создание контроллера и представлений

Поскольку с моделями мы закончили, перейдем к наиболее важной части приложения - к контроллеру. Для контроллеров у нас уже имеется папка **Controllers**. Нажмите на нее правой кнопкой мыши и в появившемся меню выберите **Add->Controller...**. Перед вами появится диалоговое окно создания нового контроллера. Позже в соответствующей главе мы подробно поговорим о параметрах создания контроллерах да и о самих контроллерах, а пока установите в качестве имени контроллера **HomeController** и нажмите Add (Добавить)



У вас будет создан следующий контроллер:

|  |  |
| --- | --- |
|  | using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.Web.Mvc;    namespace BookStore.Controllers  {      public class HomeController : Controller      {          //          // GET: /Home/          public ActionResult Index()          {              return View();          }      }  } |

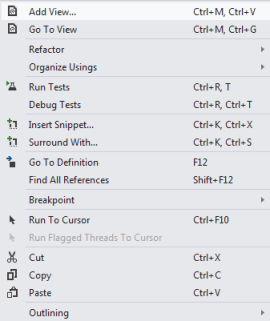
Контроллер по сути и есть главное звено приложения, которая связывает модель и пользовательский интерфейс. Обратите внимание, что контроллер - это обычный класс, который наследуется от базового класса **Controller**. Он пока имеет единственный метод **Index**, который возвращает некоторый результат метода View() - будущее представление. И если мы сейчас запустим приложение, то получим ошибку о том, что представление для данного действия не определено. Но перед тем как добавить представление и приступить к созданию пользовательского интерфейса изменим метод Index следующим образом:

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.Web.Mvc;  using BookStore.Models;    namespace BookStore.Controllers  {      public class HomeController : Controller      {          // создаем контекст данных          BookContext db = new BookContext();            public ActionResult Index()          {              // получаем из бд все объекты Book              IEnumerable<Book> books = db.Books;              // передаем все полученный объекты в динамическое свойство Books в ViewBag              ViewBag.Books = books;              // возвращаем представление              return View();          }      }  } |

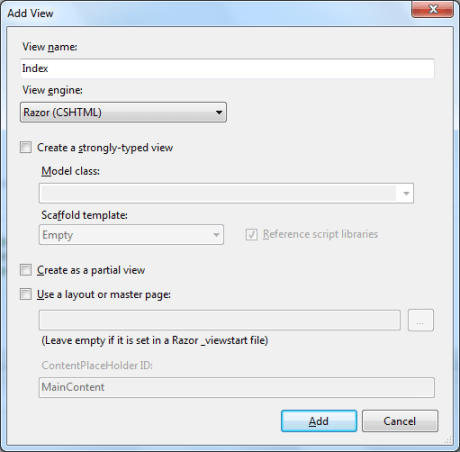
Прежде всего, обратите внимание, что, так как у нас модели находятся в другом пространстве имен, хотя и в одном проекте, то мы его должны импортировать. Затем мы создаем контекст данных. Используя свойство **Books**, мы получаем из базы данных список объектов Book.

Далее мы создаем объект Books в объекте ViewBag и присваиваем ему этот список. Объект **ViewBag** является таким объектом, который передается в представление. И мы можем определить в этом объекте любую переменную и передать ей некоторое значение, а затем в представлении извлечь это значение.

А теперь создадим само представление. Наведем курсор на метод Index и нажмем правой кнопкой мыши. Нам отобразится меню, в котором выберем **Add View...**.



Затем нам откроется окно добавления представления, где нам будет предложено выбрать имя представления и еще ряд параметров. Поскольку у нас еще нет мастер-страниц в проекте, снимем галочку с флажка **Use a layout or masterpage:** и нажмем Add.



Позже в соответствующей главе мы подробнее поговорим про опции создания и вообще про представления. А пока Visual Studio автоматически откроет созданное представление **Index.cshtml**. По сути оно пусто и содержит минимальный код разметки на языке html. Изменим его следующим образом:

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

Здесь мы создаем таблицу, в которой будут располагаться наши данные о программах кредитования. Особый интерес представляет конструкция @foreach (var c in ViewBag.Books). Эта конструкция использует синтаксис движка представления Razor (а именно его мы выбрали при создании представления). Подробнее мы погорим о движке Razor в отдельной главе, а пока вам надо знать, что после символа@ согласно синтаксису мы можем использовать выражения кода на языке C#.

В цикле мы пробегаемся по элементам в объекте ViewBag.Books, который мы ранее создали в методе контроллера. И затем получаем свойства каждого элемента и помещаем его в ячейку таблицы.

Ссылка в конце - <a href="/Home/Buy/@b.Id">Купить</a> будет означать адрес, по которому будет размещаться форма оформления покупки выбранной книги.

#### Основы маршрутизации

Чтобы вызвать контроллер HomeController или отправить ему запрос, нам надо указать в строке запроса его имя - **Home**. Кроме того, после имени контроллера нам надо через слеш указать действие или метод контроллера, к которому отправляется запрос. По умолчанию при запуске проекта или при обращении к сайту система mvc будет вызывать действие Index контроллера HomeController, если мы не укажем иной маршрут по умолчанию в параметрах маршрутизации. Путь /Home/Buy/@b.Id означает, что мы будем обращаться к методу Buy контроллера HomeController. Но перед тем как создать этот метод, наполним приложение данными.

#### Данные для моделей по умолчанию

Мы можем уже сейчас запустить наше приложение, однако кроме строки с заголовками в таблице мы ничего не увидим, потому что у нас же еще нет данных. Здесь мы могли поступить двумя способами: либо добавить в базу данных mssql некоторые данные, либо инициализировать начальные данные для моделей в коде. Выберем второй вариант. Для этого в папку Models добавим новый класс**BookDbInitializer** и добавим в него следующий код:

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.Data.Entity;    namespace BookStore.Models  {      public class BookDbInitializer : DropCreateDatabaseAlways<BookContext>      {          protected override void Seed(BookContext db)          {              db.Books.Add(new Book { Name = "Война и мир", Author = "Л. Толстой", Price = 220 });              db.Books.Add(new Book { Name = "Отцы и дети", Author = "И. Тургенев", Price = 180 });              db.Books.Add(new Book { Name = "Чайка", Author = "А. Чехов", Price = 150 });                base.Seed(db);          }      }  } |

С помощью данного класса мы создаем три объекта Book и добавляем их в базу данных. Чтобы этот класс заработал, нам надо не только его объявить, но и запустить. Все начальные настройки и конфигурации у нас хранятся в файле **Global.asax**. Откроем его и добавим в него в метод **Application\_Start**, который отрабатывает при старте приложения, следующую строчку:

|  |  |
| --- | --- |
|  | Database.SetInitializer(new BookDbInitializer()); |

Ну и конечно, импортируем в файле Global.asax пространства имен [имя\_проекта].Models и System.Data.Entity, иначе класс BookDbInitializer будет недоступен.

Все, теперь, если мы запустим приложение, то увидим, что в таблице содержатся ранее определенные нами данные:

И если мы откроем папку проекта и зайдем в ней в каталог **App\_Data**, то найдем только что созданную базу данных Bookstore.mdf, которая и будет хранить таблицы с данными.

Да, наша таблица выглядит немного неказисто, не мешало бы применить к ней стили, но не сейчас. Сейчас нам надо создать метод **Buy**, который будет отвечать за обработку ввода пользователя при оформлении покупки. Добавим в контроллер HomeController следующие два метода:

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

По сути здесь мы определили одно действие Buy, однако в одном случае оно выполняется при получении запроса GET, а во втором случае - при получении запроса POST, что мы и определили с помощью атрибутов [HttpGet] и [HttpPost]. Метод public ActionResult Buy(int id)очень прост - он просто принимает id выбранной книги и возвращает для нее соответствующее представление.

Метод public string Buy(Purchase purchase) выглядит несколько сложнее. Он принимает переданную ему в запросе POST модель purchase и добавляет ее в базу данных. В конце мы возвращаем строку сообщения.

И в конце добавим представление **Buy**. Для этого нажмем на метод public ActionResult Buy(int id) правой кнопкой и добавим в проект новое представление. Изменим код нового представления следующим образом:

|  |
| --- |
| @{      Layout = null;  }    <!DOCTYPE html>  <html>  <head>      <meta name="viewport" content="width=device-width" />      <title>Покупка</title>  </head>  <body>      <div>          <h3>Форма оформления покупки</h3>          <form method="post">              <input type="hidden" value="@ViewBag.BookId" name="BookId" />              <table>                  <tr><td><p>Введите свое имя </p></td>                      <td><input type="text" name="Person" /> </td></tr>                  <tr><td><p>Введите адрес :</p></td><td>                     <input type="text" name="Address" /> </td></tr>                  <tr><td><input type="submit" value="Отправить" /> </td><td></td></tr>              </table>          </form>      </div>  </body>  </html> |

Здесь мы создали форму ввода данных. При переходе по ссылке "/Home/Buy/2" контроллер будет получать запрос к действию Buy. И так как такой запрос представляет тип GET, пользователю будет возвращаться форма. После заполнения формы и нажатия на кнопку форма будет оправляться запросом POST, так как мы его определили в строке <form method="post" action="">. Контроллер снова будет получать запрос к методу Buy, только теперь будет выбираться для обработки запроса метод public string Buy(Purchase purchase).

Каким образом система MVC угадывает, что мы передали с запросом post информацию о модели? Обратите внимание на поля ввода <input type="text" name="Person" /> и <input type="text" name="Address" />. Их свойства **name** соответствуют именам свойств модели Purchase. При нажатии кнопки и отправки запроса передаются значения этих полей. Система MVC, используя соглашения по умолчанию, считает эти значения значениями соответствующих свойств модели.

Теперь первое мини-тренировочное приложение готово, и мы можем его запустить. Перейдите на страницу оформления заказа, заполните поля и нажмите кнопку "Отправить".

После этого ваша заявка на покупку книги попадет в базу данных, а в браузере отобразится соответствующее уведомление.

А теперь добавим к нашему сайту элементы стилизации в виде стилей css, а также мастер-страницы, которые позволяют придать всем страницам сайта единообразный вид.

## Стилизация приложения

Теперь в нашем приложении не хватает одной детали - стилизации. Сделаем какую-нибудь простенькую стилизацию. Для этого нам надо определить файл стилей для нашего приложения. Все файлы стилей принято хранить в папке Content, поэтому вначале добавим в проект папку Content. Для этого нажмем на название проекта правой кнопкой мыши и в появившемся меню выберем **Add (Добавить)->New Folder (Новая папка)**.

Теперь в новую папку добавим файл стилей. Назовем его Site.css. Для этого нажмем правой кнопкой мыши на папку Content и в появившемся меню выберем **Add (Добавить)->New Item (Новый элемент)**. Затем в списке шаблонов найдем шаблон **Style Sheet** и дадим новому файлу имя Site.css.

Теперь добавим в файл Site.css следующее содержание:

|  |
| --- |
| body {      font-size: 13px;      font-family: Verdana, Arial, Helvetica, Sans-Serif;      background-color: #f7f7fa;      padding-left:40px;  }    nav{      display: block;  }    .menu {      padding-left:10px;  }  .menu ul {      list-style:none;  }  .menu li {      display:inline;  }  .menu a:hover {      color:red;  }  table {     vertical-align:middle;     text-align:left;   }  .header {      font-weight:bold;  }   td {      padding-right:10px;   }  input {      width: 150px;  } |

Класс .menu в данном случае будет служить в качестве класса для навигационного меню на сайте.

Теперь мы можем использовать стили. Поскольку наши представления по сути обычные веб-страницы, мы можем как и на любой странице html добавить в секции head ссылки на файл стилей:

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

Если в приложении одно представление, или если разные представления используют разные стили и другие ресурсы, то такой подход оправдан. Однако в нашем случае у нас есть два представления, у которых должно быть одно меню и которые должны быть одинаково стилизованы. В этом случае очень удобно использовать мастер-страницы. Мастер-страница задает некий единый шаблон для других использующих его представлений.

Итак, создадим мастер-страницу. По сути это простое представление. Вначале добавим в папку Views проекта новую папку Shared. После этого в папку Shared добавим новое представление. Назовем его **\_Layout.cshtml**:

Добавим в файл \_Layout.cshtml следующий текст:

|  |
| --- |
| <!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> |

Здесь мы использовали другой способ подключения стилей - мы использовали метод Url.Content, в который передали путь к файлу. Оба способа - и этот, и использованный ранее равноправны.

Затем мы создали меню сайта с помощью методов Html.ActionLink, который генерирует элемент-ссылку и принимает название ссылки, метод контроллера и имя контроллера. В конце идет метод **RenderBody()**, при вызове ресурса в эту часть мастер-страницы и будут подставляться содержимое остальных представлений.

Теперь адаптируем представления к использованию их мастер-страницей. Изменим представление Index.cshtml следующим образом:

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

Также изменим представление Buy.cshtml:

|  |
| --- |
| @{      Layout = "~/Views/Shared/\_Layout.cshtml";  }        <div>          <h3>Форма оформления покупки</h3>          <form method="post" action="">              <input type="hidden" value="@ViewBag.BookId" name="BookId" />              <table>                  <tr><td><p>Введите свое имя </p></td>                      <td><input type="text" name="Person" /> </td></tr>                  <tr><td><p>Введите адрес :</p></td><td>                     <input type="text" name="Address" /> </td></tr>                  <tr><td><input type="submit" value="Отправить" /> </td>                  <td></td></tr>              </table>          </form>      </div> |

Сущность изменений состоит в том, что в секции Layout = "~/Views/Shared/\_Layout.cshtml"; мы указываем путь к мастер-странице. После этого нам не нужны секции head и body. Запустив приложение, мы увидим, что теперь к нашему сайту применена стилизация:

# Контроллеры

## Основы контроллеров

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

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

Чтобы обратиться контроллеру из веб-браузера, нам надо в адресной строке набрать **адрес\_сайта/Имя\_контроллера/**. Так, по запросу**адрес\_сайта/Home/** система маршрутизации по умолчанию вызовет метод Index контроллера HomeController для обработки входящего запроса. Если мы хотим отправить запрос к конкретному методу контроллера, то нужно указывать этот метод явно: **адрес\_сайта/Имя\_контроллера/Метод\_контроллера**, например, *адрес\_сайта/Home/Buy* - обращение к методу Buy контроллера HomeController.

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

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

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

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

|  |
| --- |
| using System.Web.Mvc;  using System.Web.Routing;    namespace BookStore.Controllers  {      public class MyController : 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:

|  |
| --- |
| using System.Web.Mvc;    namespace BookStore.Controllers  {      public class BookShopController : Controller      {          public ActionResult Index()          {              return View();          }      }  } |

## Методы действий

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

|  |
| --- |
| public class HomeController : Controller  {      BookContext db = new BookContext();        public ActionResult Index()      {          IEnumerable<Book> books = db.Books;          ViewBag.Books = books;          return View();      }        [HttpGet]      public ActionResult Buy(int id)      {          ViewBag.BookId = id;          return View();      }      [HttpPost]      public string Buy(Purchase purchase)      {          purchase.Date = DateTime.Now;          db.Purchases.Add(purchase);          db.SaveChanges();          return "Спасибо, " + purchase.Person + ", за покупку!";      }  } |

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

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

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | [HttpPost]  public string Buy(Purchase purchase)  {      purchase.Date = getToday();      db.Purchases.Add(purchase);      db.SaveChanges();      return "Спасибо, " + purchase.Person + ", за покупку!";  }  private DateTime getToday()  {      return DateTime.Now;  } |

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

## Получение входных данных

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

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

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

Кроме POST-запросов у нас есть также GET-запросы, при которых все параметры передаются в строке запроса. Например, вторая версия метода Buy в качестве параметра принимает значение типа int: public ActionResult Buy(int id). Стандартный get-запрос принимает примерно следующую форму: *название\_ресурса?параметр1=значение1&параметр2=значение2*. То есть запрос к данному методу мог бы выглядеть так: *Home/Buy?id=2*. Название параметров метода должно совпадать с названием параметров в строке запроса. Благодаря этому система сможет их автоматически связать. Ну а в самом методе мы сможем получить этот параметр и использовать его по своему усмотрению.

Кроме того, система маршрутизации позволяет создавать маршруты. Например, по умолчанию в проекте MVC определяется следующий маршрут: *Контроллер/Метод/id*. Последний параметр является опциональным. И благодаря этому мы можем передать параметр id и так:*Home/Buy/2*

Для примера определим действие, которое будет подсчитывать площадь треугольника:

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

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

Мы также можем задать для параметров значения по умолчанию:

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

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

#### Получение данных из контекста запроса

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

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

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

## Результаты действий

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

В прошлой теме в примере с вычислением площади треугольника мы возвращали html-код в виде строки. Но, как правило, возвращаемым результатом является объект класса, производного от **ActionResult**. ActionResult представляет собой абстрактный класс, в котором определен один метод **ExecuteResult**, переопределяемый в классах-наследниках:

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

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

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

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

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

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

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

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

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

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

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

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

**Встроенные классы, производные от ActionResult**

В реальности нам вряд ли потребуется часто создавать свои классы для обработки результата действия. 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 файлов
* **FileContentResult**: класс, производный от FileResult, пишет в ответ массив байтов
* **FilePathResult**: также производный от FileResult класс, пишет в ответ файл, находящийся по заданному пути
* **FileStreamResult**: класс, производный от FileResult, пишет бинарный поток в выходной ответ
* **HttpStatusCodeResult**: результат действия, который возвращает клиенту определенный статусный код HTTP
* **HttpUnauthorizedResult**: класс, производный от HttpStatusCodeResult. Возвращает клиенту ответ в виде статусного кода HTTP 401, указывая, что пользователь не прошел авторизацию и не имеет прав доступа к запрошенному ресурсу.
* **HttpNotFoundResult**: производный от HttpStatusCodeResult. Возвращает клиенту ответ в виде статусного кода HTTP 404, указывая, что запрошенный ресурс не найден
* **JavaScriptResult**: возвращает в ответ в качестве содержимого код JavaScript
* **JsonResult**: возвращает в качестве ответа объект или набор объектов в формате JSON
* **PartialViewResult**: производит рендеринг частичного представления в выходной поток
* **RedirectResult**: перенаправляет пользователя по другому адресу URL, возвращая статусный код 302 для временной переадресации или код 301 для постоянной переадресации зависимости от того, установлен ли флаг Permanent.
* **RedirectToRouteResult**: класс работает подобно RedirectResult, но перенаправляет пользователя по определенному адресу URL, указанному через параметры маршрута
* **ViewResult**: производит рендеринг представления и отправляет результаты рендеринга в виде html-страницы клиенту

Рассмотрим подробнее работу некоторых из этих классов.

## ViewResult

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 ViewResult SomeMethod()      {          return View("Index");      }  } |

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

|  |
| --- |
| public class HomeController : Controller  {      public ViewResult SomeMethod()      {          return View("~/Views/Some/Index.cshtml");      }  } |

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

**Передача данных из контроллера в представление**

Существует несколько способов передачи данных из контроллера в представление. Первый из них представляет использование объекта**ViewData**.

ViewData представляет словарь из пар ключ-значение:

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

Тогда в представлении SomeView.cshtml мы могли бы 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()  {      IEnumerable<Book> books = db.Books;      ViewBag.Books = books;      return View();  } |

Хотя ViewData и ViewBag и похожи, в то же время они не полностью эквивалентны. Так, например, нельзя передавать динамические значения из ViewBag в методы расширения в представлениях. Например, мы не можем написать @Html.TextBox("name", ViewBag.Name), так как компилятор C# должен знать тип каждого параметра во время компиляции, чтобы выбрать нужный метод расширения. В этом случае нам надо либо использовать ViewData:@Html.TextBox("name", ViewData["Name"]), либо применить приведение типов: @Html.TextBox("name", (string)ViewBag.Name)

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

## Переадресация и отправка кодов статуса и ошибок

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

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

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

Для временной переадресации применяется метод **Redirect**:

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

Для постоянной переадресации подобным образом используется метод RedirectPermanent:

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

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

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

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

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

Метод RedirectToRoute позволяет произвести перенаправление по определенному маршруту внутри домена:

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

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

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

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

**Отправка ошибок и статусных кодов**

Иногда возникает необходимость отправить сообщения об ошибках при доступе к тому или иному ресурсу. Обычно, если ресурс недоступен, mvc-фреймворк автоматически отреагирует на эту ситуацию, отправив соответствующий статусный код. Но в некоторых ситуациях нам нужно более тонко реагировать на полученный запрос. Например, у нас есть контент, к которому установлены возрастные ограничения. Мы смотрим введенный возраст, и если он попадает под ограничение, мы можем выслать статусный код ошибки:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | 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**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public ActionResult Check(int age)  {      if (age < 21)      {          return new HttpUnauthorizedResult();      }      return View();  } |

## Отправка файлов в ASP.NET MVC

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

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

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

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public FileResult GetFile()  {      // Путь к файлу      string file\_path = Server.MapPath("~/Files/PDFIcon.pdf");      // Тип файла - content-type      string file\_type="application/pdf";      // Имя файла - необязательно      string file\_name = "PDFIcon.pdf";      return File(file\_path,file\_type,file\_name);  } |

Предполагается, что у нас в проекте есть папка Files, в которой лежит файл PDFIcon.pdf. Метод **Server.MapPath** позволяет построить полный путь к ресурсу из каталога в проекте. Но также можно использовать и абсолютные пути, обращаясь к любому файлу в файловой системе, например, string file\_path = @"C:\Book\PDFIcon.pdf";

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

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | // Отправка массива байтов  public FileResult GetBytes()  {      string path = Server.MapPath("~/Files/PDFIcon.pdf");      byte[] mas = System.IO.File.ReadAllBytes(path);      string file\_type = "application/pdf";      string file\_name = "PDFIcon.pdf";      return File(mas, file\_type, file\_name);  } |

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | // Отправка потока  public FileResult GetStream()  {      string path = Server.MapPath("~/Files/PDFIcon.pdf");      // Объект Stream      FileStream fs = new FileStream(path, FileMode.Open);      string file\_type = "application/pdf";      string file\_name = "PDFIcon.pdf";      return File(fs, file\_type, file\_name);  } |

## HttpContext. Контекст запроса. Куки

Те, кто работал с php, вероятно, знают такой объект как $\_SERVER и ряд переменных, с помощью которых можно получить данные о запросе: о браузере, об ip-адресе пользователя и еще ряд других. ASP.NET MVC также предоставляет подобную функциональность, инкапсулированную в объекте **HttpContext**.

Возможно, вас может смутить тот факт, что также в контроллере вы можете обратиться к объекту ControllerContext, который имеет свойство HttpContext и по сути предоставляет доступ к той же функциональности. В то же время между ними есть некоторые различия. Объект HttpContext описывает данные конкретного http-запроса, который обрабатывается приложением. А ControllerContext описывает данные http-запроса по отношению к данному контроллеру.

Информация о запросе содержится в свойстве Request. **HttpContext.Request** представляет объект класса, унаследованного отHttpRequestBase, и поэтому содержит все его свойства. Рассмотрим некоторые из них.

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

Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public string Index()  {      string browser = HttpContext.Request.Browser.Browser;      string user\_agent = HttpContext.Request.UserAgent;      string url = HttpContext.Request.RawUrl;      string ip = HttpContext.Request.UserHostAddress;      string referrer = HttpContext.Request.UrlReferrer == null ? "" : HttpContext.Request.UrlReferrer.AbsoluteUri;      return "<p>Browser: " + browser+"</p><p>User-Agent: "+user\_agent+"</p><p>Url запроса: "+url+          "</p><p>Реферер: " + referrer + "</p><p>IP-адрес: "+ip+"</p>";  } |

HttpContext.Request имеет еще ряд свойств, которые можно посмотреть в документации.

**Работа с куки**

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

|  |  |
| --- | --- |
| 1 | string id =HttpContext.Request.Cookies["id"].Value; |

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

Однако прежде чем получать значения куки, их естественно надо установить. Для этого нам служит другое свойство - свойствоHttpContext.Response. Например, установим в куки значение "id":

|  |  |
| --- | --- |
| 1 | HttpContext.Response.Cookies["id"].Value = "ca-4353w"; |

**Отправка ответа**

Свойство HttpContext.Response содержит объект HttpResponse, который передает на сторону клиента некоторые значения. Например, куки, как было описано выше. Также мы можем передать с помощью этого свойства другие объекты, например, кодировку ответа:HttpContext.Response.Charset = "iso-8859-2";

Мы также можем управлять ответом с помощью методов. К примеру метод AddHeader позволяет добавить к ответу дополнительный заголовок.

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public string ContextData()  {      HttpContext.Response.Write("<h1>Hello World</h1>");        string user\_agent = HttpContext.Request.UserAgent;      string url = HttpContext.Request.RawUrl;      string ip = HttpContext.Request.UserHostAddress;      string referrer = HttpContext.Request.UrlReferrer == null ? "" : HttpContext.Request.UrlReferrer.AbsoluteUri;      return "<p>User-Agent: "+user\_agent+"</p><p>Url запроса: "+url+          "</p><p>Реферер: " + referrer + "</p><p>IP-адрес: "+ip+"</p>";  } |

Либо так:

|  |  |
| --- | --- |
| 1  2  3  4 | public void ContextData()  {      HttpContext.Response.Write("<h1>Hello World</h1>");  } |

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

**Определение пользователя**

Также объект HttpContext позволяет определить пользователя:

|  |  |
| --- | --- |
| 1  2  3 | bool IsAdmin = HttpContext.User.IsInRole("admin"); // определяем, принадлежит ли пользователь к администраторам  bool IsAuth=HttpContext.User.Identity.IsAuthenticated; // аутентифицирован ли пользователь  string login = HttpContext.User.Identity.Name; // логин авторизованного пользователя |

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

## Асинхронные контроллеры и методы

Уже в версиях MVC 2 и MVC 3 появились асинхронные контроллеры, однако с появлением **Task Parallel Library** работа с асинхронными контроллерами была предельно упрощена.

Асинхронные контроллеры служат для обработки таких запросов, которые занимают или могут занять довольно продолжительное время, например, обращение к базе данных для получения большой порции данных. Применение асинхронных контроллеров позволяет приложению продолжать выполнять другие запросы, параллельно выполняя асинхронный код. А использование модификаторов **async** и **await** позволяет создавать асинхронные методы, призванные выполнять продолжительные операции без блокирования основного потока.

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

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

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

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

Сравним на примере вызов синхронного и асинхронного метода:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.Web.Mvc;  using AsyncContollers.Models;  using System.Threading.Tasks;  using System.Data.Entity;    namespace AsyncContollers.Controllers  {      public class HomeController : Controller      {          BookContext db = new BookContext();            public ActionResult Index()          {              IEnumerable<Book> books = db.Books;              ViewBag.Books = books;              return View();          }            // асинхронный метод          public async Task<ActionResult> BookList()          {              IEnumerable<Book> books = await Task.Run(() => db.Books);              ViewBag.Books = books;              return View("Index");          }      }  } |

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

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

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

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

Но также следует учитывать, что await используется с методами, возвращающими объект Task. Поэтому получение данных их БД мы обернули в такое выражение await Task.Run(() => db.Books);. Метод Task.Run в качестве параметра получает некоторое действие в виде лямбда-выражения. А этим действием и является извлечение данных из БД: db.Books.

Вместо метода Task.Run мы также могли использовать метод Task.Factory.StartNew. Кроме того, мы можем использовать в одном асинхронном методе несколько модификаторов await для получения результатов нескольких асинхронных задач.

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

# Представления

## Введение в представления

Хотя работа приложения MVC управляется главным образом контроллерами, но непосредственно пользователю приложение доступно в виде представления, которое и формирует внешний вид приложения. В ASP.NET MVC 4 представления представляют файлы с расширением*cshtml/vbhtml/aspx/ascx*, которые содержат код с интерфейсом пользователя, как правило, на языке html. Стандартное представление:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | @{      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> |

Несмотря на то, что представление в основном состоит из кода html, оно не является html-страницей. При компиляции приложения на основе требуемого представления сначала генерируется класс на языке C#, а затем этот класс компилируется. Например, из выше приведенного представления будет генерироваться примерно в такой класс:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91 | #pragma checksum "c:\users\hp\documents\visual studio 2013\Projects\MVC\BookStore\BookStore\Views\Home\SomeView.cshtml" "{ff1816ec-aa5e-4d10-87f7-6f4963833460}" "1F05F4D370C9D00F8CBDFB8BD1F51D74189D0617"    namespace ASP {      using System;      using System.Collections.Generic;      using System.IO;      using System.Linq;      using System.Net;      using System.Web;      using System.Web.Helpers;      using System.Web.Security;      using System.Web.UI;      using System.Web.WebPages;      using System.Web.Mvc;      using System.Web.Mvc.Ajax;      using System.Web.Mvc.Html;      using System.Web.Optimization;      using System.Web.Routing;      using BookStore;        public class \_Page\_Views\_Home\_SomeView\_cshtml : System.Web.Mvc.WebViewPage<dynamic> {    #line hidden            public \_Page\_Views\_Home\_SomeView\_cshtml() {          }            protected ASP.global\_asax ApplicationInstance {              get {                  return ((ASP.global\_asax)(Context.ApplicationInstance));              }          }            public override void Execute() {  BeginContext("~/Views/Home/SomeView.cshtml", 0, 2, true);    WriteLiteral("\r\n");    EndContext("~/Views/Home/SomeView.cshtml", 0, 2, true);                  #line 2 "c:\users\hp\documents\visual studio 2013\Projects\MVC\BookStore\BookStore\Views\Home\SomeView.cshtml"        Layout = null;                  #line default              #line hidden  BeginContext("~/Views/Home/SomeView.cshtml", 27, 48, true);    WriteLiteral("\r\n\r\n<!DOCTYPE html>\r\n\r\n<html>\r\n<head>\r\n    <meta");    EndContext("~/Views/Home/SomeView.cshtml", 27, 48, true);    BeginContext("~/Views/Home/SomeView.cshtml", 75, 16, true);    WriteLiteral(" name=\"viewport\"");    EndContext("~/Views/Home/SomeView.cshtml", 75, 16, true);    BeginContext("~/Views/Home/SomeView.cshtml", 91, 29, true);    WriteLiteral(" content=\"width=device-width\"");    EndContext("~/Views/Home/SomeView.cshtml", 91, 29, true);    BeginContext("~/Views/Home/SomeView.cshtml", 120, 74, true);    WriteLiteral(" />\r\n    <title>SomeView</title>\r\n</head>\r\n<body>\r\n    <div> \r\n       <h2>");    EndContext("~/Views/Home/SomeView.cshtml", 120, 74, true);    BeginContext("~/Views/Home/SomeView.cshtml", 195, 15, false);                  #line 15 "c:\users\hp\documents\visual studio 2013\Projects\MVC\BookStore\BookStore\Views\Home\SomeView.cshtml"        Write(ViewBag.Message);                  #line default              #line hidden  EndContext("~/Views/Home/SomeView.cshtml", 195, 15, false);    BeginContext("~/Views/Home/SomeView.cshtml", 210, 38, true);    WriteLiteral("</h2> \r\n    </div>\r\n</body>\r\n</html>\r\n");    EndContext("~/Views/Home/SomeView.cshtml", 210, 38, true);          }      }  } |

Хотя код не самый читабельный, особенно если не знать, что делают все эти классы и методы, но здесь мы можем увидеть, что при компиляции создается класс, наследующий от класса System.Web.Mvc.WebViewPage<T>, где T - это класс модели, которая будет использоваться. Но так как представление не строго типизированное, поэтому вместо имени класса модели идет ключевое слово dynamic. Все действия данного класса заключены в методе Execute, в котором с помощью метода WriteLiteral обрабатываются все имеющиеся в представлении элементы разметки html.

Найти сгенерированные из представлений файлы кода можно по пути *C:\Users\Имя\_Логина\AppData\Local\Temp\Temporary ASP.NET Files\root*. Правда, все папки имеют зашифрованные имена, поэтому чтобы определить нужную папку приложения, нужно будет затратить некоторое время на поиск. Кроме того, сами генерируемые файлы кода также имеют зашифрованные имена. Так, выше приведенный класс в моем случеае имеет имя *App\_Web\_nri53fza.1.cs* и находится в папке *root\307f1c1d\a36bbd4f*.

### 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:

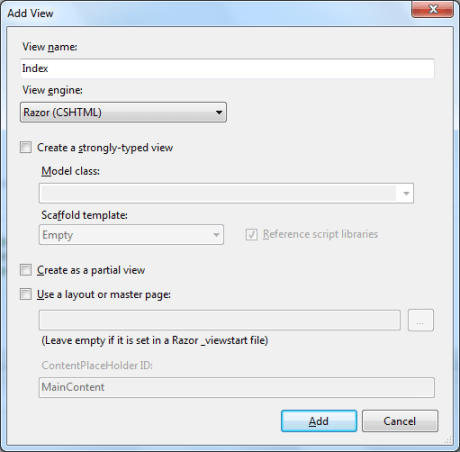
|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public ActionResult Index()  {      IEnumerable<Book> books = db.Books;      ViewBag.Books = books;      return View();  } |

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public ActionResult Index()  {      IEnumerable<Book> books = db.Books;      ViewBag.Books = books;      return View("~/Views/Some/SomeView.cshtml");  } |

### Создание нового представления в MVC 4

При создании представления нам предлагается целый ряд настроек:



Это диалоговое окно имеет следующие опции:

* **View Engine** - выпадающий список для выбора движка представления. С ASP.NET MVC 4 данное диалоговое окно поддерживает несколько движков представлений. По умолчанию здесь представлены две возможности для выбора: Razor и ASPX. Но при установке сторонних движков они также могут быть доступны для выбора.
* **Create a strongly-typed view** - эта опция позволит создать строго типизированное представление. После ее выбора нужно будет указать модель в выпадающем списке, который содержит список всех моделей. Но чтобы все модели приложения можно было использовать, перед добавлением представления нужно скомпилировать проект.
* **Scaffold template** - опция позволяет выбрать шаблон формирования нового представления.

##### Типы шаблонов представления

* + **Empty**. Создается пустое представление. В представлении только определен тип модели с помощью директивы @model
  + **Create**. Создается представление с формой для создания новых объектов модели. Генерируется метка и поле редактирования для каждого свойства модели.
  + **Delete**. Создается представление с формой для удаления существующих объектов модели. Отображаются метка и текущее значение каждого свойства модели
  + **Details**. Создается представление, которое отображает метку и значение каждого свойства модели
  + **Edit**. Создается представление с формой для редактирования существующих объектов модели. Генерируется метка и поле редактирования для каждого свойства модели.
  + **List**. Создается представление с таблицей всех моделей. Столбцы отображают свойства модели. В данное представление вы должны передавать из метода контроллера значение типа IEnumerable<Тип\_модели>. Представление также содержит ссылки на методы для выполнения операций создания/правки/удаления.
* **Reference Script Libraries** - эта опция показывает, будет ли ваше представление включать набор файлов JavaScript. По умолчанию файл \_Layout.cshtml содержит ссылку на главную библиотеку jQuery, однако не имеет ссылки на библиотеки jQuery Validation и Unobtrusive jQuery Validation.

При создании представления, которое содержит форму ввода данных, как например, шаблоны представлений Edit или Create, установка данной опции гарантирует, что представление будет включать ссылки на вышеупомянутые библиотеки. Данные библиотеки необходимы для валидации ввода пользователя на стороне клиента. Во всех других случаях вы можете проигнорировать эту опцию.

* **Create as a Partial View** - выбор этой опции указывает, что созданное представление будет неполным. В итоге представление будет вполне обычным, однако в его шапке не будет таких тегов, как <html> и <head>
* **Use a layout or Master Page** - эта опция указывает, будут ли использоваться мастер-страницы (главные страницы) или представление будет самодостаточным. 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.

## Строго типизированные представления

В предыдущих примерах, чтобы передать информацию из БД в представление, мы использовали объект ViewBag:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @foreach (var b in ViewBag.Books)  {      <tr>          <td><p>@b.Name</p></td>          <td><p>@b.Author</p></td>          <td><p>@b.Price</p></td>          <td><p><a href="/Home/Buy/@b.Id">Купить</a></p></td>      </tr>  } |

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @foreach (BookStore.Models.Book b in ViewBag.Books)  {      <tr>          <td><p>@b.Name</p></td>          <td><p>@b.Author</p></td>          <td><p>@b.Price</p></td>          <td><p><a href="/Home/Buy/@b.Id">Купить</a></p></td>      </tr>  } |

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | BookContext db = new BookContext();    public ActionResult Index()  {      return View(db.Books);  } |

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

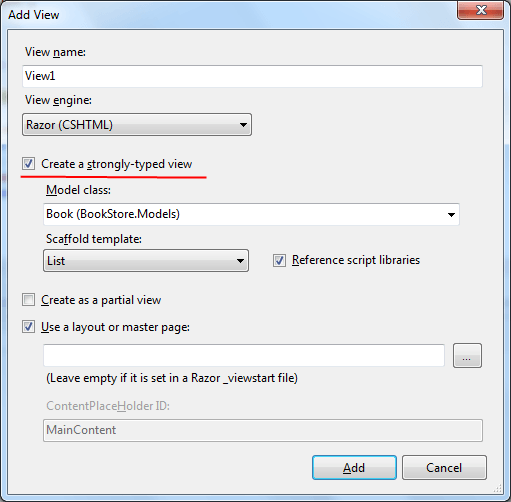
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | @model IEnumerable<BookStore.Models.Book>  @{      Layout = "~/Views/Shared/\_Layout.cshtml";  }      <div>          <h3>Распродажа книг</h3>          <table>              <tr class="header"><td><p>Название книги</p></td>                  <td><p>Автор</p></td>                  <td><p>Цена</p></td><td></td>              </tr>              @foreach (BookStore.Models.Book b in Model)              {              <tr>                  <td><p>@b.Name</p></td>                  <td><p>@b.Author</p></td>                  <td><p>@b.Price</p></td>                  <td><p><a href="/Home/Buy/@b.Id">Купить</a></p></td>              </tr>              }          </table>      </div> |

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

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

|  |  |
| --- | --- |
| 1  2  3 | @using BookStore.Models  @model IEnumerable<Book>  .............................. |

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



В поле модели укажем какой-нибудь имеющийся класс модели, а в поле шаблонов выберем любой шаблон, кроме Empty. И если мы, к примеру, выберем шаблон List, то автоматически сгенерированное представление будет по своему функционалу идентично ранее рассмотренному представлению с выводом книг.

## Движок представлений и Razor

При использовании элементов кода С# в представлениях мы использовали знак @. Это знак, используемый движком представлений Razor для перехода к коду на языке C#/VB.NET. Чтобы понять суть движка Razor и его синтаксиса, посмотрим сначала, что такое движки представлений.

### Движок представлений

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

До MVC 3 использовался только движок **Web Forms**, который использовал стандартный синтаксис ASP.NET. Например:

|  |  |
| --- | --- |
| 1  2  3 | <% foreach(BookStore.Models.Book b in ViewBag.Books) { %>      <li>Книга: <%: b.Name %></li>  <% } %> |

Введение в MVC 3 движка Razor позволило уменьшить синтаксис при вызове кода C#, сделать код более "чистым". Аналогичный пример на Razor выгдядит так:@foreach(BookStore.Models.Book b in ViewBag.Books) {<li>Книга: b.Name </li>} Поэтому его использование более предпочтительно. В конечном счете, выбор движка представлений зависит от личных предпочтений, и оба движка по функциональности равноценны.

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

Кроме движка Razor можно использовать и движки сторонних производителей. Например, движок [Spark](http://sparkviewengine.com/), использующий декларативный синтаксис. Или [Brail](http://mvccontrib.org/), который использует язык Boo.

#### Основы синтаксиса Razor

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

Например, переход к выражению кода:

|  |  |
| --- | --- |
| 1 | <p>@b.Name</p> |

Razor автоматически распознает, что Name - это свойство объекта b.

Также можно использовать стандартные классы и методы, например, выведем текущее время:

|  |  |
| --- | --- |
| 1 | <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:

|  |  |
| --- | --- |
| 1  2  3  4 | @foreach (BookStore.Models.Book b in Model)  {      <p>@b.Name</p>  } |

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

|  |  |
| --- | --- |
| 1  2  3  4  5 | @{      string head = "Привет мир!!!";      head = head + " Добро пожаловать на сайт!";  }   <h3>@head</h3> |

## Мастер-страницы

Для создания единообразного вида сайта применяются мастер-страницы. Мастер-страницы - это по сути те же самые представления. Мы можем определить на мастер-странице некоторые элементы, которые будут отображаться на всех страницах сайта. А также мы можем определить заполнители или плейсхолдеры, содержание которых обеспечивают другие представления.

При создании первого приложения в предыдущих главах мы использовали следующую мастер-страницу \_Layout.chtml:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | <!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 вид это обычное представление за одним исключением - вызова метода @RenderBody(). Этот вызов является плейсхолдером, на место которого другие представления, которые используют эту мастер-страницу, будут подставлять свое содержимое. И таким образом, мы можем легко установить для представлений веб-приложения единообразный стиль.

Чтобы применить мастер-страницу к представлению, мы должны в секции **Layout** указать путь к мастер-странице. Например, наше представление Index.cshtml, использующее эту мастер-страницу, начинается так:

|  |  |
| --- | --- |
| 1  2  3 | @{      Layout = "~/Views/Shared/\_Layout.cshtml";  } |

Если мы не используем мастер-страницу, то мы указываем Layout = null;.

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | <!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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @{      Layout = "~/Views/Shared/\_Layout.cshtml";  }  <!-- здесь остальное содержание -->  @section Footer {      Все права защищены. Syte Corp. 2012.  } |

Однако, при таком подходе, если у нас есть куча представлений, и мы вдруг захотели определить новую секцию на мастер-странице, нам придется изменить все имеющиеся представления, что не очень удобно. Но мастер-страницы предлагают нам варианты 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:

|  |  |
| --- | --- |
| 1 | <footer>@RenderSection("Footer", false)</footer> |

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <footer>      @if (IsSectionDefined("Footer")) {          @RenderSection("Footer")      }      else      {          <span>Содержание элемента footer по умолчанию.</span>      }  </footer> |

**ViewStart**

Если у нас в проекте пара-тройка представлений, мы легко можем изменить для каждого описание мастер-страницы в секции Layout, если, например, мы решим использовать другую мастер-страницу. Однако, 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 3/4 по шаблону Basic или Internet в каталоге Views уже находится файл \_ViewStart.cshtml. Этот файл определяет мастер-страницу, используемую по умолчанию. При создании нового проекта ASP.NET MVC 3/4 по шаблону Empty, нам надо добавить новое представление **\_ViewStart.cshtml** в папку Views и удалить весь ранее автоматически сгенерированный текст и добавить туда следующие строки:

|  |  |
| --- | --- |
| 1  2  3 | @{      Layout = "~/Views/Shared/\_Layout.cshtml";  } |

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

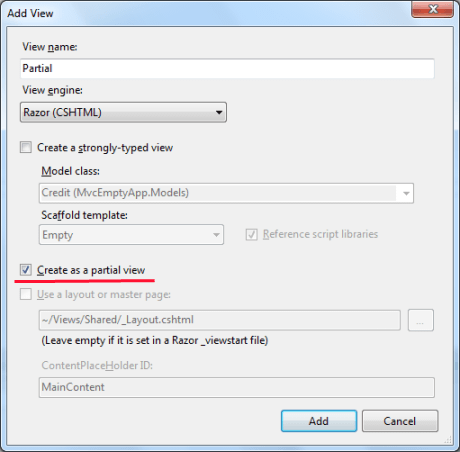
## Частичные представления

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

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

|  |  |
| --- | --- |
| 1  2  3  4  5 | public ActionResult Partial()  {      ViewBag.Message = "Это частичное представление.";      return PartialView();  } |

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



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

<h2>@ViewBag.Message</h2>

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

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

@Html.Partial("Partial")

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

|  |  |
| --- | --- |
| 1  2  3  4  5 | public ActionResult Index()  {      ViewBag.Message = "Это вызов частичного представления из обычного";      return View();  } |

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

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

@{Html.RenderPartial("Partial");}

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

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @model IEnumerable<string>  <h2>Список стран</h2>  <ul>  @foreach (string t in Model)  {      <li>@t</li>  }  </ul> |

Тогда мы можем вызвать это представление так: @Html.Partial("Partial", new string[] { "Russia", "USA", "Canada", "France" })

## HTML-хелперы

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

**Строчные хелперы**

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @helper BookList(IEnumerable<BookStore.Models.Book> books)  {      <ul>          @foreach (BookStore.Models.Book b in books)          {              <li>@b.Name</li>          }      </ul>  } |

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

|  |  |
| --- | --- |
| 1  2  3  4 | <h3>Список книг</h3>  @BookList(ViewBag.Books)  <!-- или если используется строго типизированное представление -->  @BookList(Model) |

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | @helper CreateList(string[] all)  {      <ul>          @foreach (string s in all)          {              <li>@s</li>          }      </ul>  }  @{      string[] cities = new string[] { "Лондон", "Париж", "Москва" };  }  @{      string[] countries = new string[] { "Великобритания", "Франция", "Россия" };  }  <h3>Города</h3>  @CreateList(cities)  <br />  <h3>Страны</h3>  @CreateList(countries) |

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

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | using System;  using System.Web;  using System.Web.Mvc;  using System.Linq;    namespace BookStore.Helpers  {      public static class ListHelper      {          public static MvcHtmlString CreateList(this HtmlHelper html, string[] items)          {              TagBuilder ul = new TagBuilder("ul");              foreach (string item in items)              {                  TagBuilder li = new TagBuilder("li");                  li.SetInnerText(item);                  ul.InnerHtml += li.ToString();              }              return new MvcHtmlString(ul.ToString());          }      }  } |

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

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

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

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

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | @{      string[] cities = new string[] { "Лондон", "Париж", "Москва" };  }  @{      string[] countries = new string[] { "Великобритания", "Франция", "Россия" };  }  @using BookStore.Helpers  <h3>Города</h3>  @Html.CreateList(cities)  <br />  <h3>Страны</h3>  <!-- или можно вызвать так -->  @ListHelper.CreateList(Html, countries) |

## Работа с формами

Хотя мы можем сами написать любой требуемый хелпер, но фреймворк MVC уже предоставляет большой набор встроенных html-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.

### Хелпер Html.BeginForm

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <form method="post" action="/Home/Buy">      <input type="hidden" value="@ViewBag.BookId" name="BookId" />      <table>          <tr><td><p>Введите свое имя </p></td>              <td><input type="text" name="Person" /> </td></tr>          <tr><td><p>Введите адрес :</p></td>              <td><input type="text" name="Address" /> </td></tr>          <tr><td><input type="submit" value="Отправить" /> </td>                  <td></td></tr>      </table>  </form> |

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | @using(Html.BeginForm("Buy", "Home", FormMethod.Post))  {      <input type="hidden" value="@ViewBag.BookId" name="BookId" />      <table>          <tr><td><p>Введите свое имя </p></td>              <td><input type="text" name="Person" /> </td></tr>          <tr><td><p>Введите адрес :</p></td>              <td><input type="text" name="Address" /> </td></tr>          <tr><td><input type="submit" value="Отправить" /> </td>                  <td></td></tr>      </table>  } |

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

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | [HttpGet]  public ActionResult Buy()  {      return View();  }    [HttpPost]  public string Buy(Purchase purchase)  {      ..............      return "Спасибо за покупку книги";;  } |

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

|  |  |
| --- | --- |
| 1  2  3  4 | @using(Html.BeginForm())  {      .............  } |

### Ввод информации

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

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

#### Html.TextBox

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @using(Html.BeginForm("Buy", "Home", FormMethod.Post))  {      <input type="hidden" value="@ViewBag.BookId" name="BookId" />      <p>Введите свое имя: </p>      @Html.TextBox("Person", "Введите имя")      <p>Введите адрес :</p>      @Html.TextBox("Address", "Введите адрес")      <p><input type="submit" value="Отправить" /></p>  } |

Мы получим тот же результат:

#### Html.TextArea

Хелпер TextArea используется для создания элемента <textarea>, который представляет многострочное текстовое поле. Результатом выражения @Html.TextArea("text", "привет <br/> мир")

будет следующая html-разметка:

|  |  |
| --- | --- |
| 1  2 | <textarea cols="20" id="text" name="text" rows="2">привет <br/> мир  </textarea> |

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

|  |  |
| --- | --- |
| 1 | @Html.TextArea("text", "привет <br /> мир", 5, 50, null) |

Этот хелпер сгенерирует следующую разметку:

|  |  |
| --- | --- |
| 1  2 | <textarea cols="50" id="text" name="text" rows="5">привет <br /> мир  </textarea> |

#### Html.Hidden

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

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

сгенерирует разметку:

|  |  |
| --- | --- |
| 1 | <input id="BookId" name="BookId" type="hidden" value="2" /> |

А при передачи 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:

|  |  |
| --- | --- |
| 1 | <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):

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <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:

|  |  |
| --- | --- |
| 1  2 | <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:

|  |  |
| --- | --- |
| 1 | <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 преобразует их в последовательность объектов SelectListItem. Так, код@Html.DropDownList("countires", new SelectList(new string[] {"Russia","USA", "Canada","France"}),"Countries") генерирует следующую разметку:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <select id="countires" name="countires"><option value="">Countries</option>  <option>Russia</option>  <option>USA</option>  <option>Canada</option>  <option>France</option>  </select> |

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | BookContext db = new BookContext();    public ActionResult Index()  {      SelectList books = new SelectList(db.Books, "Author", "Name");      ViewBag.Books = books;      return View();  } |

Здесь мы создаем объект SelectList, передавая в его конструктор набор значений для списка (db.Books), название свойства модели Book, которое будет использоваться в качестве значения (Author), и название свойства модели Book, которое будет использоваться для отображения в списке. В данном случае необязательно устанавливать два разных свойства, можно было установить одно и то же свойство и для значения, и для отображения.

Тогда в представлении мы можем так использовать этот SelectList:

|  |  |
| --- | --- |
| 1 | @Html.DropDownList("Author", ViewBag.Books as SelectList) |

И при рендеринге представления все элементы SelectList добавятся в выпадающий список

#### Html.ListBox

Хелпер Html.ListBox, также как и DropDownList, создает элемент <select />, но при этом делает возможным множественное выделение элементов (то есть для атрибута multiple устанавливается значение multiple). Для создания списка, поддерживающего множественное выделение, вместо SelectList можно использовать класс MultiSelectList:

|  |  |
| --- | --- |
| 1 | @Html.ListBox("countires", new MultiSelectList(new string[] {"Россия","США", "Китай","Индия"})) |

Этот код генерирует следующую разметку:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | [HttpPost]  public string Index(string[] countries)  {      string result = "";      foreach (string c in countries)      {          result += c;          result += ";";      }      return "Вы выбрали: " + result;  } |

## Строго типизированные хелперы

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

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class Purchase  {      public int PurchaseId { get; set; }      public string Person { get; set; }      public string Address { get; set; }      public int BookId { get; set; }      public DateTime Date { get; set; }  } |

И для ее использования применялась следующая форма:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <form method="post" action="">      <input type="hidden" value="@ViewBag.BookId" name="BookId" />          <table>              <tr><td><p>Введите свое имя </p></td>                      <td><input type="text" name="Person" /> </td></tr>              <tr><td><p>Введите адрес :</p></td><td>                     <input type="text" name="Address" /> </td></tr>              <tr><td><input type="submit" value="Отправить" /> </td>                      <td></td></tr>      </table>  </form> |

Перепишем этот пример с использованием хелперов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | @model BookStore.Models.Purchase  @{      Layout = "~/Views/Shared/\_Layout.cshtml";  }        <div>          <h3>Форма оформления покупки</h3>          @using(Html.BeginForm("Buy", "Home", FormMethod.Post))          {              @Html.HiddenFor(m=>m.BookId)              @Html.LabelFor(m => m.Person,"Введите свое имя")              <br />              @Html.TextBoxFor(m=>m.Person)              <br /><br />              @Html.LabelFor(m => m.Address, "Введите адрес")              <br />              @Html.TextBoxFor(m=>m.Address)              <p><input type="submit" value="Отправить" /></p>          }      </div> |

Строго типизированный хелпер похож на обычный, только в конце прибавляется суффикс **For**: LabelFor. Так как строго типизированные хелперы могут использоваться только в строго типизированных представлениях, то вначале представления указываем модель, которая будет использоваться: @model BookStore.Models.Purchase. То есть, в вызове @Html.TextBoxFor(m=>m.Person) параметр m представляет переменную модели Purchase. А лямбда-выражение m=>m.Person указывает, что данный хелпер будет генерировать текстовое поле для свойства Person. Таким образом, хелпер @Html.TextBoxFor(m=>m.Person) сгенерирует текстовое поле <input id="Person" name="Person" type="text" value="" />.

Для каждого базового встроенного хелпера имеется свой строго типизированный хелпер:

* **Html.CheckBoxFor**

Выражение @Html.CheckBoxFor(m=>m.Enable, false) cоздает разметку:

|  |  |
| --- | --- |
| 1  2 | <input id="Enable" name="Enable" type="checkbox" value="true" />  <input name="Enable" type="hidden" value="false" /> |

* **Html.HiddenFor**

Выражение @Html.HiddenFor(m=> m.Name) создает разметку:

<input id="Name" name="Name" type="hidden" value="значение m.Name" />

* **Html.LabelFor**

Хелпер @Html.LabelFor(m => m.Name,"Имя") генерирует разметку:

<p<<label for="Name">Имя</label>

</p<

* **Html.PasswordFor**

Хелпер @Html.PasswordFor(m => m.Password) оздает разметку:

<p<<input id="Password" name="Password" type="password" />

</p<

* **Html.RadioButtonFor**

@Html.RadioButtonFor(m => m.Option, "val") генерирует разметку:

<input id="Option" name="Option" type="radio" value="val" />

* **Html.TextBoxFor**

Выражение

@Html.TextBoxFor(m => m.Name) создает разметку:

<input id="Name" name="Name" type="text" />

**Html.TextAreaFor**

Хелпер @Html.TextAreaFor(m => m.Name,10,9,null) генерирует код:

<textarea cols="9" id="Name" name="Name" rows="10" ></textarea>

# Модели

## Модели и БД

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

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | public class Book  {      // ID книги      public int Id { get; set; }      // название книги      public string Name { get; set; }      // автор книги      public string Author { get; set; }      // цена      public int Price { get; set; }  }  public class Purchase  {      // ID покупки      public int PurchaseId { get; set; }      // имя и фамилия покупателя      public string Person { get; set; }      // адрес покупателя      public string Address { get; set; }      // ID книги      public int BookId { get; set; }      // дата покупки      public DateTime Date { get; set; }  } |

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

Данные моделей хранятся в базе данных. Чтобы взаимодействовать с базой данных, очень удобно пользоваться фреймворком **Entity Framework**. 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.

Выпуск ASP.NET MVC 4 уже включает Entity Framework 5.0, однако в проектах по типу Empty вам придется подключать фреймворк через пакетный менеджер NuGet.

Чтобы подключиться к базе данных через Entity Framework, нам нужен **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. Допустим, создадим контекст данных для работы с вышеприведенными моделями Book и Purchase:

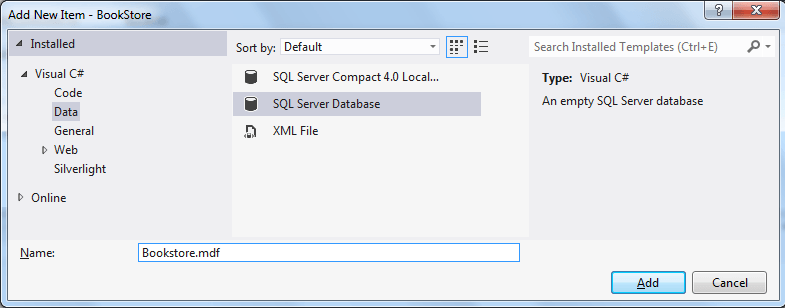
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | using System;  using System.Collections.Generic;  using System.Web;  using System.Data.Entity;    namespace BookStore.Models  {      public class BookContext : DbContext      {          public DbSet<Book> Books { get; set; }          public DbSet<Purchase> Purchases { get; set; }      }  } |

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

## Подключение к базе данных

Чтобы хранить данные, нам естественным образом нужна база данных. Как правило, в качестве базы данных используется 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 у нас предназначена папка **App\_Data**. Для этого нажмем правой кнопкой мыши на папку App\_Data и в появившемся контекстном меню выберем**Add-> New Item...**. В появившемся окне добавления нового элемента выберем **SQL Server Database** и назовем новую базу данных Bookstore.mdf:



Мы можем создать базу данных равнозначным образом и на сервере. После этого база данных добавляется в проект, и мы можем увидеть ее в папке App\_Data. Теперь в обозревателе баз данных (окно **Database Explorer**) мы можем подключиться к ней и создать таблицы, которые будут хранить данные.

Раскроем узел Bookstore.mdf и найдем узел Tables. Нажмем на этот узел правой кнопкой мыши и в появившемся меню выберем пункт *Add New Table*. И перед нами появится окно, в котором нам надо определить названия и типы столбцов новой таблицы. По соглашениям о наименованиях таблицы при работе с Entity Framework должны соответствовать имени модели. То есть, так как наша модель называетсяBook, то таблица будет называться Books. А Entity Framework автоматически распознает, что таблица Books соответствует классу Book.

Итак, создадим структуру таблицы:

Не забудьте установить ниже в окне Properties (в Visual Stidio 2010 - окно Column Properties) для столбца Id соответствующие параметры для первичного ключа:

После этого, если мы работаем с Visual Studio 2010, нам будет предложено просто ввести имя таблицы - введем имя Books, и затем таблица добавляется в БД.

А в Visual Studio Express 2012 for Web нам надо сгенерировать таблицу на основе заданного выше определения. Для этого нажмем на кнопку Update:

В появившемся диалоговом окне нажмем на кнопку Update Database. После этого в нашу базу данных добавляется только что сгенерированная таблица. Подобным образом определим таблицу Purchases для модели Purchase:

Добавим в таблицу Books несколько записей:

Теперь, во-первых, чтобы взаимодействовать с БД, нам нужен класс контекста данных, пусть это будет следующий класс BookContext:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.Data.Entity;    namespace BookStore.Models  {      public class BookContext : DbContext      {          public DbSet<Book> Books { get; set; }          public DbSet<Purchase> Purchases { get; set; }      }  } |

Во-вторых, определим строку подключения к БД. Для этого откроем файл Web.config и добавим в конец секции **configuration** определение строки подключения. Однако тут надо сразу заметить, что для Visual Studio 2010 строка подключения будет отличаться от строки подключения, которая используется в Visual Studio 2012.

Итак, строка подключения для Visual Studio 2010 будет выглядеть так:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <configuration>  .........................      <connectionStrings>          <add name="BookContext" connectionString="Data Source=.\SQLEXPRESS;AttachDbFilename='|DataDirectory|\Bookstore.mdf';Integrated Security=True;User Instance=True"   providerName="System.Data.SqlClient" />      </connectionStrings>  </configuration> |

Для Visual Studio 2012 будет выглядеть определение строки подключения будет выглядеть следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <configuration>  .........................      <connectionStrings>          <add name="BookContext" connectionString="Data Source=(LocalDB)\v11.0;AttachDbFilename='|DataDirectory|\Bookstore.mdf';Integrated Security=True"   providerName="System.Data.SqlClient" />      </connectionStrings>  </configuration> |

В Visual Studio 2012 в отличие от 2010-й версии мы можем использовать режим LocalDB, который предназначен прежде всего для разработчиков, представляя некоторую упрощенную версию. Поэтому в данном случае в качестве источника данных используется**(LocalDB)\v11.0**.

Обратите внимание, что в обоих случаях свойство name="BookContext" должно содержать название контекста данных.

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

Теперь мы можем получить содержимое таблицы Books в контроллере Home:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public class HomeController : Controller  {      BookContext db = new BookContext();        public ActionResult Index()      {          return View(db.Books);      }  } |

И вывести данные в представлении Index.cshtml:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | @model IEnumerable<BookStore.Models.Book>  @{      Layout = "~/Views/Shared/\_Layout.cshtml";  }      <div>          <h3>Распродажа книг</h3>          <table>              <tr class="header"><td><p>Название книги</p></td>                  <td><p>Автор</p></td>                  <td><p>Цена</p></td><td></td>              </tr>              @foreach (BookStore.Models.Book b in Model)              {              <tr>                  <td><p>@b.Name</p></td>                  <td><p>@b.Author</p></td>                  <td><p>@b.Price</p></td>                  <td><p><a href="/Home/Buy/@b.Id">Купить</a></p></td>              </tr>              }          </table>      </div> |

**Закрытие подключения**

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

|  |  |
| --- | --- |
| 1  2  3  4  5 | protected override void Dispose(bool disposing)  {      db.Dispose();      base.Dispose(disposing);  } |

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

## Шаблонные хелперы

Кроме стандартных html-хелперов, рассмотренных в прошлой главе и генерирующих определенные элементы разметки html, фреймворк MVC также имеет **шаблонные (также называют шаблонизированные) хелперы**. Зачем они нужны? Они более гибки по сравнению с html-хелперами, так как в этом случае нам не надо думать, какой нам надо создать элемент разметки и какой для этого выбрать хелпер. Мы просто сообщаем шаблонному хелперу, какое свойство модели мы хотим использовать, а фреймворк уже сам выбирает, какой html-элемент сгенерировать, исходя из типа свойства и его метаданных.

Шаблонные хелперы:

* **Display**

Создает элемент разметки, который доступен только для чтения, для указанного свойства модели: Html.Display("Name")

* **DisplayFor**

Строго типизированный аналог хелпера Display: Html.DisplayFor(e => e.Name)

* **Editor**

Создает элемент разметки, который доступен для редактирования, для указанного свойства модели: Html.Editor("Name")

* **EditorFor**

Строго типизированный аналог хелпера Editor: Html.EditorFor(e => e.Name)

* **DisplayText**

Создает выражение для указанного свойства модели в виде простой строки: Html.DisplayText("Name")

* **DisplayTextFor**

Строго типизированный аналог хелпера DisplayText: Html.DisplayTextFor(e => e.Name)

Кроме данных шаблонов, которые используются для отдельного свойства модели, есть еще несколько шаблонов, которые позволяют сгенерировать разом все поля для определенной модели:

* **DisplayForModel**

Создает поля для чтения для всех свойств модели: Html.DisplayForModel()

* **EditorForModel**

Создает поля для редактирования для всех свойств модели: Html.EditorForModel()

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public ActionResult BookView(int id)  {      var book = \_db.Books.Find(id);      if (book != null)      {          return View(book);      }      return RedirectToAction("Index");  } |

Теперь создадим представление BookView.cshtml, в котором все поля генерируются одним хелпером:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @{      Layout = "~/Views/Shared/\_Layout.cshtml";  }    @model BookStore.Models.Book    <h2>Книга № @Model.Id</h2>  @Html.DisplayForModel() |

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

## Редактирование модели

В прошлом разделе мы увидели, что для редактирования полей модели удобно применять хелпер Editor/EditorFor. Теперь посмотрим, как сделать саму логику редактирования модели. Пусть в некотором действии контроллера мы получаем объект модели по Id и выводим ее поля для редактирования в представлении:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | [HttpGet]  public ActionResult EditBook(int? id)  {      if (id == null)      {          return HttpNotFound();      }      Book book = db.Books.Find(id);        if (book == null)      {          return HttpNotFound();      }      return View(book);  } |

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

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | @{      ViewBag.Title = "Редактировать книгу";      Layout = "~/Views/Shared/\_Layout.cshtml";  }  @model BookStore.Models.Book  <h2>Книга № @Model.Id</h2>  @using (Html.BeginForm("EditBook", "Home", FormMethod.Post))  {      <fieldset>          @Html.HiddenFor(m => m.Id)          <p>              @Html.LabelFor(m => m.Name, "Название книги")              <br />              @Html.EditorFor(m => m.Name)          </p>          <p>              @Html.LabelFor(m => m.Author, "Автор")              <br />              @Html.EditorFor(m => m.Author)            </p>          <p>              @Html.LabelFor(m => m.Price, "Цена")              <br />              @Html.EditorFor(m => m.Price)          </p>          <p><input type="submit" value="Отправить" /></p>      </fieldset>  } |

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

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | [HttpPost]  public ActionResult EditBook(Book book)  {      db.Entry(book).State = EntityState.Modified;      db.SaveChanges();      return RedirectToAction("Index");  } |

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

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

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

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

## Добавление и удаление модели

**Добавление модели**

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | [HttpGet]  public ActionResult Create()  {      return View();  }  [HttpPost]  public ActionResult Create(Book book)  {      db.Books.Add(book);      db.SaveChanges();        return RedirectToAction("Index");  } |

Первый метод возвращает пользователю представление с формой для добавления, а второй - принимает данные этой формы. Теперь создадим представление.

А представление будет выглядеть следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | @model BookStore.Models.Book    @{      Layout = "~/Views/Shared/\_Layout.cshtml";  }    <h2>Новая книга</h2>    @using (Html.BeginForm())  {      @Html.LabelFor(model => model.Name, "Название книги")      <br />      @Html.EditorFor(model => model.Name)      <br /><br />      @Html.LabelFor(model => model.Author, "Автор")      <br />      @Html.EditorFor(model => model.Author)      <br /><br />      @Html.LabelFor(model => model.Price, "Цена")      <br />      @Html.EditorFor(model => model.Price)      <br /><br />      <input type="submit" value="Добавить" />  } |

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | [HttpPost]  public ActionResult Create(Book book)  {      db.Entry(book).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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public ActionResult Delete(int id)  {      Book b = db.Books.Find(id);      if (b != null)      {          db.Books.Remove(b);          db.SaveChanges();      }      return RedirectToAction("Index");  } |

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public ActionResult Delete(int id)  {      Book b = new Book { Id = id };      db.Entry(b).State = EntityState.Deleted;      db.SaveChanges();        return RedirectToAction("Index");  } |

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

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

<img src="http://адрес\_нашего\_сайта/Home/Delete/1" />

В итоге при открытии письма 1-я запись в таблице может быть удалена. Уязвимость качается 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | [HttpGet]  public ActionResult Delete(int? id)  {      if (id == null)      {          return HttpNotFound();      }      Book b = db.Books.Find(id);      if (b == null)      {          return HttpNotFound();      }      return View(b);  }  [HttpPost, ActionName("Delete")]  public ActionResult DeleteConfirmed(int? id)  {      if (id == null)      {          return HttpNotFound();      }      Book b = db.Books.Find(id);      if (b == null)      {          return HttpNotFound();      }      db.Books.Remove(b);      db.SaveChanges();      return RedirectToAction("Index");  } |

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

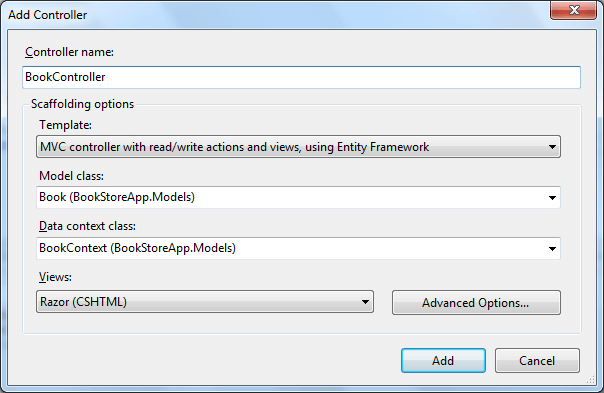
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | @{      Layout = "~/Views/Shared/\_Layout.cshtml";  }  @model BookStore.Models.Book  <h2>Удаление книги</h2>  <dl>      <dt>Название</dt>      <dd>          @Html.DisplayFor(model => model.Name)      </dd>        <dt>Автор</dt>      <dd>          @Html.DisplayFor(model => model.Author)      </dd>        <dt>Цена</dt>      <dd>          @Html.DisplayFor(model => model.Price)      </dd>  </dl>    @using (Html.BeginForm())  {      <input type="submit" value="Удалить"  />  } |

## Шаблоны формирования

Поскольку зачастую разработчики вынуждены создавать представления для одних и тех же действий: добавления, изменения, удаления и просмотра записей из БД, то команда разработчиков MVC внедрила такую полезную функцию, как **шаблоны формирования (scaffolding templates)**. Эти шаблоны позволяют по заданной модели и контексту данных сформировать всю необходимую разметку для представлений и контроллера, с помощью которых можно управлять записями в БД.

Тут надо сразу оговориться, что для корректного применения шаблонов формирования надо, чтобы использовалась одна из связок MVC 4 + EntityFramework 5 или MVC 5 + Entity Framework 6.

Итак, применим шаблон формирования. Добавим новый контроллер. В окне создания нового контроллера установим следующие параметры:



В секции **Template** нам предлагается выбрать шаблон формирования кода. По умолчанию нам доступны следующие шаблоны:

* **Empty MVC Controller**. Этот шаблон добавляет в папку Controllers класс, производный от класса Controller с указанным именем. Единственное действие, которое выполняет этот контроллер - это действие Index, которое по умолчанию не заключает никакого функционала. Данный шаблон не создает представлений
* **MVC Controller with Empty Read/Write Actions**. Этот шаблон добавляет в проект контроллер, который содержит методы Index, Details, Create, Edit и Delete. Однако эти методы фактически никакой полезной работы не выполняют, и необходимо самим создать для них код и представления для каждого из этих методов.
* **MVC Controller with Read/Write Actions and Views, Using Entity Framework**. Этот шаблон 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. Что очень полезно, так как нам не надо писать соответствующий код, как в предыдущем разделе.
* **Empty API Controller**. Этот шаблон пустой контроллер, производный от класса ApiController. Класс ApiController был введен в MVC 4 и используется в проектах типа Web API - своего рода веб-служб.
* **API Controller with Empty Read/Write Actions**. Этот шаблон добавляет в проект контроллер, производный от класса ApiController, который содержит методы Index, Details, Create, Edit и Delete.
* **API Controller with Read/Write Actions and Views, Using Entity Framework**

Этот шаблон, также как и в случае с обычным контроллером, создает контроллер, только в данном случае производный от класса ApiController, который содержит методы Index, Details, Create, Edit и Delete, а также все необходимые представления для этих действий.

В нашем случае мы выбираем пункт **MVC Controller with Read/Write Actions and Views, Using Entity Framework**.

Далее нам будет предложено выбрать в поле **Model class** модель, на основе которой будут создаваться представления. Выберем созданную ранее модель Book (либо какую-то другую имеющуюся модель).

В завершении в поле **Data context class** надо выбрать класс контекста данных, с помощью которого мы получаем все данные модели из БД. В нашем случае - это ранее созданный класс BookContext.

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120 | using System;  using System.Collections.Generic;  using System.Data;  using System.Data.Entity;  using System.Linq;  using System.Web;  using System.Web.Mvc;  using BookStoreApp.Models;    namespace BookStoreApp.Controllers  {      public class BookController : Controller      {          private BookContext db = new BookContext();            //          // GET: /Book/            public ActionResult Index()          {              return View(db.Books.ToList());          }            //          // GET: /Book/Details/5            public ActionResult Details(int id = 0)          {              Book book = db.Books.Find(id);              if (book == null)              {                  return HttpNotFound();              }              return View(book);          }            //          // GET: /Book/Create            public ActionResult Create()          {              return View();          }            //          // POST: /Book/Create            [HttpPost]          public ActionResult Create(Book book)          {              if (ModelState.IsValid)              {                  db.Books.Add(book);                  db.SaveChanges();                  return RedirectToAction("Index");              }                return View(book);          }            //          // GET: /Book/Edit/5            public ActionResult Edit(int id = 0)          {              Book book = db.Books.Find(id);              if (book == null)              {                  return HttpNotFound();              }              return View(book);          }            //          // POST: /Book/Edit/5            [HttpPost]          public ActionResult Edit(Book book)          {              if (ModelState.IsValid)              {                  db.Entry(book).State = EntityState.Modified;                  db.SaveChanges();                  return RedirectToAction("Index");              }              return View(book);          }            //          // GET: /Book/Delete/5            public ActionResult Delete(int id = 0)          {              Book book = db.Books.Find(id);              if (book == null)              {                  return HttpNotFound();              }              return View(book);          }            //          // POST: /Book/Delete/5            [HttpPost, ActionName("Delete")]          public ActionResult DeleteConfirmed(int id)          {              Book book = db.Books.Find(id);              db.Books.Remove(book);              db.SaveChanges();              return RedirectToAction("Index");          }            protected override void Dispose(bool disposing)          {              db.Dispose();              base.Dispose(disposing);          }      }  } |

А в папке *Views/Book* мы увидим все необходимые представления со всем необходимым кодом, который теперь нам не надо набирать вручную. мы можем запустить проект и перейти в адресной строке браузера к нашему контроллеру, чтобы убедиться, что все работает как надо:

Благодаря шаблонам формирования мы можем не думать о создании кода для стандартных операций. Нам остается после генерации кода лишь изменить автоматически сгенерированные названия на свои (например, название страницы, автоматические генерируемых ссылок и др.)

## Модели со сложной структурой

**П**

Ранее использованные нами модели Book и Purchase имели довольно простую структуру - одни простые свойства типа int или string. Например, рассмотрим следующие две модели:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | public class Player  {      public int Id { get; set; }      public string Name { get; set; }      public int Age { get; set; }      public string Position { get; set; }        public int? TeamId { get; set; }      public Team Team { get; set; }  }    public class Team  {      public int Id { get; set; }      public string Name { get; set; }      public string Coach { get; set; }        public IEnumerable<Player> Players { get; set; }  } |

Здесь класс Player имеет свойство Team, которое показывает принадлежность игрока к определенной команде. Свойство Team называется**навигационным свойством** - при получении данных об игроке оно автоматически будет получать данные из БД. Однако для этого нам 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 нам надо обозвать свойство TeamId, что собственно и было сделано в вышеприведенном коде.
* *Имя\_класса\_связанной\_таблицы+Имя ключа из связанной таблицы* - в нашем случае класс Team, а ключа из модели Team - Id, поэтому опять же в этом случае получается TeamId.

Теперь создадим контекст данных, использующий модели:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public class SoccerContext : DbContext  {      public DbSet<Player> Players { get; set; }      public DbSet<Team> Teams { get; set; }  } |

Теперь посмотрим, как бы это все располагалось в БД. Допустим, у нас есть некоторая база данных *SoccerInfo.mdf*. Пусть наши модели Player и Team располагаются соответственно в таблицах Players и Teams.

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

Скрипт создания таблицы:

CREATE TABLE [dbo].[Teams] (

[Id] INT NOT NULL PRIMARY KEY IDENTITY,

[Name] NVARCHAR (50) NOT NULL,

[Coach] NVARCHAR (50) NOT NULL

);

Тогда чтобы проецировать модель Player на таблицу *Players* в базе данных, нам надо задать следующее определение столбцов таблицы:

Скрипт создания таблицы:

CREATE TABLE [dbo].[Players] (

[Id] INT NOT NULL PRIMARY KEY IDENTITY,

[Name] NVARCHAR (50) NOT NULL,

[Age] INT NOT NULL,

[Position] NVARCHAR (50) NOT NULL,

[TeamId] INT NULL,

CONSTRAINT [FK\_Players\_Teams] FOREIGN KEY ([TeamId]) REFERENCES [Teams] ([Id]) ON DELETE SET NULL

);

Кроме того, здесь мы задаем внешний ключ - свойство **TeamId** теперь будет ссылаться на свойство **Id** из таблицы Teams.

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

CONSTRAINT [FK\_Players\_Teams] FOREIGN KEY ([TeamId]) REFERENCES [Teams]([Id]) ON DELETE SET NULL

Это обычное выражение языка SQL, которое связывает столбцы двух таблиц.

Последняя часть этого выражения (ON DELETE SET NULL) указывает, что при удалении объекта из таблицы Teams, свойству TeamId, которое ссылалось на удаленный объект, будет присвоено значение null.

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

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

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

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.Web.Mvc;  using NavigationProperty.Models;  using System.Data.Entity;  using System.Data;    namespace NavigationProperty.Controllers  {      public class HomeController : Controller      {          SoccerContext db = new SoccerContext();            // Выводим всех футболистов          public ActionResult Index()          {              var players = db.Players.Include(p => p.Team);              return View(players.ToList());          }      }  } |

Теперь с помощью метода Include фреймворк подгружает для каждого игрока также и команду, ассоциированную с определенным игроком. А при выводе модели в представление Index.cshtml фреймворк будет выводить для каждого игрока название команды:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | @model IEnumerable<NavigationProperty.Models.Player>  @{      ViewBag.Title = "Каталог игроков";  }    <h2>Каталог игроков</h2>  <p>      @Html.ActionLink("Добавить игрока", "Create")  </p>  <table>      <tr>          <th>Имя игрока</th>          <th>Возраст</th>          <th>Позиция на поле</th>          <th>Команда</th>          <th></th>      </tr>        @foreach (var item in Model)      {          <tr>              <td>                  @Html.DisplayFor(modelItem => item.Name)              </td>              <td>                  @Html.DisplayFor(modelItem => item.Age)              </td>              <td>                  @Html.DisplayFor(modelItem => item.Position)              </td>              <td>                  @Html.DisplayFor(modelItem => item.Team.Name)              </td>              <td>                  @Html.ActionLink("Редактировать", "Edit", new { id = item.Id }) |                  @Html.ActionLink("Удалить", "Delete", new { id = item.Id })              </td>          </tr>      }  </table>  <p>      @Html.ActionLink("Каталог команд", "ListTeams")  </p> |

Поскольку мы в контроллере с помощью метода Include получили связанный с игроком объект Team по навигационному свойству TeamId, то при передаче в контроллер всех объектов Player к ним также цепляются связанные с ними объекты Team, поэтому мы можем вполне использовать в представлении свойство item.Team.Name для получения имени команды.

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public ActionResult TeamDetails(int? id)  {      if (id == null)      {          return HttpNotFound();      }      Team team = db.Teams.Find(id);      if (team == null)      {          return HttpNotFound();      }      team.Players = db.Players.Where(m=> m.TeamId== team.Id);      return View(team);  } |

Во-первых, чтобы обработать ввод при отсутствии передаваемого значения, в качестве параметра используем int? id. Во-вторых, мы подгружаем всех игроков, связанных с командой, в выражении team.Players = db.Players.Where(m=> m.TeamId== team.Id).

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | @using NavigationProperty.Models  @model Team    @{      ViewBag.Title = "Команда " + @Model.Name;      Layout = "~/Views/Shared/\_Layout.cshtml";  }    <div>      <h4>Команда @Model.Name</h4>      <hr />      <dl>          <dt>Название</dt>            <dd>              @Html.DisplayFor(model => model.Name)          </dd>            <dt>Тренер</dt>            <dd>              @Html.DisplayFor(model => model.Coach)          </dd>            <dt>Игроки</dt>            <dd>              <ul>                  @foreach (Player player in Model.Players)                  {                      <li>@player.Name (@player.Position)</li>                  }              </ul>          </dd>      </dl>  </div> |

## Работа со сложными моделями

В прошлом разделе мы создали две модели Player и Team и вывели элементы из таблицы Players на страницу. Теперь посмотрим, как проделать остальные операции с моделями - редактирование, удаление, добавление. Эти действия не сильно отличаются от логики работы с простыми моделями. Единственное отличие - мы должны учитывать значения навигационное свойство, имеющееся в сложной модели.

**Добавление модели**

Рассмотрим логику добавления модели Player в таблицу Players. Вначале пропишем в контроллере следующее действие Create:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | [HttpGet]  public ActionResult Create()  {      // Формируем список команд для передачи в представление      SelectList teams = new SelectList(db.Teams, "Id", "Name");      ViewBag.Teams = teams;      return View();  }    [HttpPost]  public ActionResult Create(Player player)  {      //Добавляем игрока в таблицу      db.Players.Add(player);      db.SaveChanges();      // перенаправляем на главную страницу      return RedirectToAction("Index");  } |

Первый вариант действия Create обрабатывает GET-запрос и выдает представление, передавая в него объект **SelectList** - список всех команд.

Второй вариант действия получает введенную пользователем в представлении модель и добавляет ее в БД. Рассмотрим само представление *Create.cshtml*:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | @model NavigationProperty.Models.Player    @{      ViewBag.Title = "Добавление игрока";      Layout = "~/Views/Shared/\_Layout.cshtml";  }    <h2>Добавление нового игрока</h2>    @using (Html.BeginForm())  {      <fieldset>          <legend>Футболист</legend>            <p>              Имя игрока <br />              @Html.EditorFor(model => model.Name)          </p>            <p>              Возраст <br />              @Html.EditorFor(model => model.Age)          </p>            <p>              Позиция на поле  <br />              @Html.EditorFor(model => model.Position)          </p>          <p>              Команда <br />              @Html.DropDownListFor(model => model.TeamId, ViewBag.Teams as SelectList)          </p>            <p>              <input type="submit" value="Добавить игрока" />          </p>      </fieldset>  }  <div>      @Html.ActionLink("К списку игроков", "Index")  </div> |

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

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

**Редактирование модели**

Редактирование работает подобным способом. Определим в контроллере действия, которые отвечают за редактирование:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | [HttpGet]  public ActionResult Edit(int? id)  {      if (id == null)      {          return HttpNotFound();      }      // Находим в бд футболиста      Player player = db.Players.Find(id);      if (player != null)      {          // Создаем список команд для передачи в представление          SelectList teams = new SelectList(db.Teams, "Id", "Name", player.TeamId);          ViewBag.Teams = teams;          return View(player);      }      return RedirectToAction("Index");  }    [HttpPost]  public ActionResult Edit(Player player)  {      db.Entry(player).State = EntityState.Modified;      db.SaveChanges();      return RedirectToAction("Index");  } |

Здесь мы также формируем список команд, которые мы получаем из БД, в объекте SelectList. И после получения запроса на редактирования определенной модели Player контроллер передает эту модель и список команд в представление *Edit.cshtml*:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | @model NavigationProperty.Models.Player  @{      ViewBag.Title = "Edit";      Layout = "~/Views/Shared/\_Layout.cshtml";  }    <h2>Изменение игрока</h2>    @using (Html.BeginForm())  {      <fieldset>          <legend>Футболист</legend>            @Html.HiddenFor(model => model.Id)            <p> Имя игрока <br />              @Html.EditorFor(model => model.Name)          </p>            <p>Возраст <br />          @Html.EditorFor(model => model.Age)          </p>            <p>Позиция на поле  <br />              @Html.EditorFor(model => model.Position)          </p>          <p>Команда <br />              @Html.DropDownListFor(model => model.TeamId, ViewBag.Teams as SelectList)          </p>          <p>              <input type="submit" value="Сохранить" />          </p>      </fieldset>  }  <div>      @Html.ActionLink("Вернуться к списку футболистов", "Index")  </div> |

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

## Модели со связью многие-ко-многим

Кроме моделей со связью один-ко-одному и один-ко-многим, которые были рассмотрены в прошлых темах, модели по типу "многие-ко-многим". Если посмотреть на мир вокруг себя, то мы сможем найти подобные модели. Наиболее распространенный и хрестоматийный пример - учеба в университете, где различное количество студентов может посещать различное количество дисциплин. И при этом у нас может возникнуть необходимость, как вести учет студентов по конкретной дисциплине, так и вести учет различных дисциплин для конкретного студента. Попробуем смоделировать данную ситуацию в приложении ASP.NET MVC 4.

Во-первых, создадим новое приложение по шаблону Basic. Первым делом нам надо создать модели. Сразу скажу, что для простоты будем использовать подход Code First.

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public class Student  {      public int Id { get; set; }      public string Name { get; set; }      public string Surname { get; set; }        public virtual ICollection<Course> Courses { get; set; }      public Student()      {          Courses = new List<Course>();      }  } |

И также добавим модель Course:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class Course  {      public int Id { get; set; }      public string Name { get; set; }        public virtual ICollection<Student> Students { get; set; }      public Course()      {          Students = new List<Student>();      }  } |

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

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public class StudentsContext : DbContext  {      public DbSet<Student> Students { get; set; }      public DbSet<Course> Courses { get; set; }        public StudentsContext() : base("DefaultConnection")      {}        protected override void OnModelCreating(DbModelBuilder modelBuilder)      {          modelBuilder.Entity<Course>().HasMany(c => c.Students)              .WithMany(s => s.Courses)              .Map(t => t.MapLeftKey("CourseId")              .MapRightKey("StudentId")              .ToTable("CourseStudent"));      }  } |

Во-первых, чтобы использовать строку подключения по умолчанию, установим для нее контекст данных в конструкторе: public StudentsContext() : base("DefaultConnection")

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

Для построения этой таблицы мы переопределяем метод OnModelCreating, в котором с помощью объекта modelBuilder создаем новую таблицу и определяем ее поля. Одно ее поле - CourseId - будет ссылаться на таблицу Courses и хранить в себе id курса. А второе поле - StudentId - будет ссылаться на таблицу студентов и хранить id студента. В итоге у нас получится набор пар id курса - id студента, благодаря этому мы сможем определить связь многие-ко-многим.

И теперь проинициализируем базу данных начальными данными. Добавим в проект следующий класс:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | public class CourseDbInitializer : DropCreateDatabaseAlways<StudentsContext>  {      protected override void Seed(StudentsContext context)      {          Student s1 = new Student { Id = 1, Name = "Егор", Surname = "Иванов" };          Student s2 = new Student { Id = 2, Name = "Мария", Surname = "Васильева" };          Student s3 = new Student { Id = 3, Name = "Олег", Surname = "Кузнецов" };          Student s4 = new Student { Id = 4, Name = "Ольга", Surname = "Петрова" };            context.Students.Add(s1);          context.Students.Add(s2);          context.Students.Add(s3);          context.Students.Add(s4);            Course c1 = new Course { Id = 1, Name = "Операционные системы",                              Students = new List<Student>() { s1, s2, s3 } };          Course c2 = new Course { Id = 2, Name = "Алгоритмы и структуры данных",                              Students = new List<Student>() { s2, s4 } };          Course c3 = new Course { Id = 3, Name = "Основы HTML и CSS",                              Students = new List<Student>() { s3, s4, s1 } };            context.Courses.Add(c1);          context.Courses.Add(c2);          context.Courses.Add(c3);            base.Seed(context);      }  } |

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

|  |  |
| --- | --- |
| 1 | Database.SetInitializer(new CourseDbInitializer()); |

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | using System;  using System.Collections.Generic;  using System.Data;  using System.Data.Entity;  using System.Linq;  using System.Web;  using System.Web.Mvc;  using TeamMvc4.Models;    namespace TeamMvc4.Controllers  {      public class HomeController : Controller      {          private StudentsContext db = new StudentsContext();            public ActionResult Index()          {              return View(db.Students.ToList());          }            public ActionResult Details(int id = 0)          {              Student student = db.Students.Find(id);              if (student == null)              {                  return HttpNotFound();              }              return View(student);          }            protected override void Dispose(bool disposing)          {              db.Dispose();              base.Dispose(disposing);          }      }  } |

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | @using TeamMvc4.Models  @model Student  @{      ViewBag.Title = "Details";  }    <fieldset>      <legend>Информация о студенте</legend>        <div class="display-label"><b>Имя</b></div>      <div class="display-field">          @Html.DisplayFor(model => model.Name)      </div>        <div class="display-label"><b>Фамилия</b></div>      <div class="display-field">          @Html.DisplayFor(model => model.Surname)      </div>        <div class="display-label"><b>Курсы</b></div>      <ul>          @foreach (Course c in Model.Courses)          {              <li>@c.Name</li>          }      </ul>    </fieldset> |

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

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

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

## Работа с моделями со связью многие-ко-многим

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

В итоге форма редактирования у нас будет выглядеть примерно так:

Итак, добавим в контроллер HomeController следующее действие Edit:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | public ActionResult Edit(int id = 0)  {      Student student = db.Students.Find(id);      if (student == null)      {          return HttpNotFound();      }      ViewBag.Courses = db.Courses.ToList();      return View(student);  }    [HttpPost]  public ActionResult Edit(Student student, int[] selectedCourses)  {      Student newStudent = db.Students.Find(student.Id);      newStudent.Name = student.Name;      newStudent.Surname = student.Surname;        newStudent.Courses.Clear();      if (selectedCourses != null)      {          //получаем выбранные курсы          foreach (var c in db.Courses.Where(co => selectedCourses.Contains(co.Id)))          {              newStudent.Courses.Add(c);          }      }        db.Entry(newStudent).State = EntityState.Modified;      db.SaveChanges();      return RedirectToAction("Index");  } |

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

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

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

И напоследок добавим само представление *Edit.cshtml*:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | @using TeamMvc4.Models  @model Student  @{      ViewBag.Title = "Edit";  }    @using (Html.BeginForm()) {        <fieldset>          <legend>Студент</legend>            @Html.HiddenFor(model => model.Id)            <div class="editor-label"><b>Имя</b></div>          <div class="editor-field">              @Html.EditorFor(model => model.Name)          </div>            <div class="editor-label"><b>Фамилия</b></div>          <div class="editor-field">              @Html.EditorFor(model => model.Surname)          </div>          <div class="editor-label"><b>Курсы</b></div>          @{              List<Course> courses = ViewBag.Courses;              foreach (Course c in courses)              {                  <input type="checkbox" name="selectedCourses" value="@c.Id"                 @(Model.Courses.Contains(c) ? "checked=\"checked\"" : "") />@c.Name <br />              }          }            <p>              <input type="submit" value="Сохранить" />          </p>      </fieldset>  } |

Ну а сохранение модели будет во многом идентично редактированию.

## Передача данных в контроллер

Ранее мы посмотрели, как передавать отдельные единичные данные и отдельные объекты моделей в методы действия контроллера в качестве параметров. Однако в реальности может возникнуть задача, передавать в метод не один элемент типа int или один объект какого-то класса, а несколько объектов. Давайте разберем возможные случаи.

**Передача коллекции**

Определим следующую форму в представлении:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @using (Html.BeginForm())  {      @Html.TextBox("names")      @Html.TextBox("names")      @Html.TextBox("names")      @Html.TextBox("names")      <input type="submit" />  } |

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <form action="/Home/Array" method="post">  <input id="names" name="names" type="text" value="" />  <input id="names" name="names" type="text" value="" />  <input id="names" name="names" type="text" value="" />  <input id="names" name="names" type="text" value="" />      <input type="submit" />  </form> |

И при отправке формы у нас будет формироваться коллекция из names, состоящая из четырех элементов. И в действии контроллера мы сможем получить все эти элементы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | [HttpPost]  public string Array(List<string> names)  {      string fin="";      for (int i = 0; i < names.Count; i++)      {          fin += names[i] + ";  ";      }      return fin;  } |

**Передача коллекции объектов модели**

Допустим, мы передаем в представление массив объектов некоторой модели в представление, например, модели Book:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | [HttpGet]  public ActionResult Add()  {      var allbooks = db.Books.ToList<Book>();        return View(allbooks);  } |

Чтобы вывести объекты для редактирования в представление мы можем использовать следующую конструкцию:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | @model List<Mvc4BasicApplication.Models.Book>  .......................................  @using (Html.BeginForm())  {      for(int i=0; i<Model.Count; i++)      {          <h4>Книга № @(i+1)</h4>            @: Name: @Html.EditorFor(m=>m[i].Name)          @: Author: @Html.EditorFor(m => m[i].Author)          @: Year: @Html.EditorFor(m => m[i].Year)      }      <input type="submit" />  } |

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

|  |  |
| --- | --- |
| 1  2  3  4  5 | [HttpPost]  public string Add(List<Book> books)  {      //........................  } |

**Передача разных объектов модели**

В предыдущем случае мы передавали коллекцию объектов модели Book. Но у нас может возникнуть ситуация, когда мы должны разграничить как-то переданные объекты. Например, метод контроллера выглядит так:

|  |  |
| --- | --- |
| 1  2  3  4  5 | [HttpPost]  public string Add(Book book, Book myBook)  {      //........................  } |

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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | @using Mvc4BasicApplication.Models  @model Book  @{      ViewBag.Title = "Add";  }  @{      Book myBook = new Book() { Name = "Мартин Иден", Author = "Джек Лондон", Year = 1910 };      }  <h2>Книги</h2>    @using (Html.BeginForm())  {      @Html.EditorFor(m=>myBook)      @Html.EditorForModel()      <input type="submit" />  } |

Одну модель мы передаем из контроллера в представление:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | [HttpGet]  public ActionResult Add()  {      Book firstBook = db.Books.ToList<Book>().First();      return View(firstBook);  } |

Вторую модель - myBook мы создаем в представлении. Все поля для модели мы создаем с помощью хелпера @Html.EditorFor(m=>myBook). Обратите внимание на имя модели - **myBook**. Поскольку мы ожидаем, что данная модель должна передаваться в качестве параметра myBookв метод, то она должна иметь точно такое же имя, а не произвольное.

Таким образом, мы можем передать на сервер два разных объекта одной модели.

# Маршрутизация

## Определение маршрутов

В предыдущих главах при обращении к некоторому действию контроллера мы набирали в браузере адрес наподобие следующего*http://localhost/Home/Index*, где *Home* являлся именем контроллера без префикса *Controller*, а *Index* - именем действия этого контроллера. Если метод Index принимал какой-нибудь параметр, например, типа int: public ActionResult Index(int Id), то мы могли обратиться к этому методу и передать значение в его параметр с помощью следующей строки: *http://localhost/Home/Index/5*. Но мы не говорили еще о том, почему мы должны прописывать маршрут именно так, и как мы собственно можем управлять маршрутами.

Посмотрим, как определен маршрут. Если в MVC 3 для определения маршрута по умолчанию в файле **Global.asax.cs** создавался специальный метод RegisterRoutes, который определял маршрут по умолчанию и потом вызывался в методе Application\_Start:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | using System.Web.Routing;  using System.Data.Entity;    namespace MvcEmptyApp  {      public class MvcApplication : System.Web.HttpApplication      {          protected void Application\_Start()          {              RegisterRoutes(RouteTable.Routes);              .............................          }            public static void RegisterRoutes(RouteCollection routes)          {              //Здесь определение маршрутов              ................................          }      }  } |

То в MVC 4 все начальные настройки конфигурации для файла Global.asax.cs вынесены в классы, расположенные в папке **App\_Start**. И затем эти классы вызываются в файле Global.asax.cs:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | using System.Web;  using System.Web.Http;  using System.Web.Mvc;  using System.Web.Routing;    namespace MvcEmptyApp  {      public class MvcApplication : System.Web.HttpApplication      {          protected void Application\_Start()          {              AreaRegistration.RegisterAllAreas();                FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);              RouteConfig.RegisterRoutes(RouteTable.Routes);          }      }  } |

Откроем файл **RouteConfig.cs**, расположенный в папке App\_Start, в котором и находится определение маршрута по умолчанию:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.Web.Http;  using System.Web.Mvc;  using System.Web.Routing;    namespace MvcEmptyApp  {      public class RouteConfig      {          public static void RegisterRoutes(RouteCollection routes)          {              routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");                routes.MapHttpRoute(                  name: "DefaultApi",                  routeTemplate: "api/{controller}/{id}",                  defaults: new { id = RouteParameter.Optional }              );                routes.MapRoute(                  name: "Default",                  url: "{controller}/{action}/{id}",                  defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }              );          }      }  } |

Цель первой строки routes.IgnoreRoute("{resource}.axd/{\*pathInfo}"); отключить обработку запросов для некоторых файлов, например с расширением \*.axd (WebResource.axd). Следующие два вызова - routes.MapHttpRoute и routes.MapRoute как раз и задают определение маршрута. Главное отличие состоит в том, что вызов routes.MapHttpRoute устанавливает сопоставления запроса некоторому маршруту для ресурса Web API, а routes.MapRoute устанавливает маршрут для обычного контроллера, которые мы использовали в предыдущих главах.

Поскольку тему Web API мы рассмотрим чуть позднее, поэтому пока поговорим об определении маршрута на примере обычного контроллера, к тому же в обоих определениях маршрутов много похожего.

Итак, метод MapRoute выполняет сопоставление маршрута запросу. Он имеет ряд перегруженных версий, которые помогают указать параметры сопоставления.

Здесь мы сначала задаем имя маршрута с помощью свойства name (в данном случае имя *Default*). С помощью параметра url мы задаем**шаблон Url**, с которым будет сопоставляться данный маршрут.

Шаблон URL состоит из нескольких сегментов, заключенных в фигурные скобки (сегмент - это часть запроса, находящаяся между слешами, но не включающая эти слеши). Каждый сегмент шаблона содержит параметр. Эти параметры называются параметрами URL.

При этом именовать параметры можно как угодно, используя любые алфавитно-цифровые символы. При получении запроса механизм маршрутизации парсит строку URL и помещает значения маршрута в словарь - в объект RouteValueDictionary, доступный через контекст приложения RequestContext. В качестве ключей используются имена параметров URL, а соответствующие сегменты URL в качестве значений. То есть, если строка запроса URL выглядит следующим образом: *http://localhost/Home/Index/5*, то у нас образуются следующие пары ключей и значений в словаре RouteValueDictionary:

|  |  |
| --- | --- |
| **Параметр** | **Значение** |
| controller | Home |
| action | Index |
| id | 5 |

Следующий параметр - defaults определяет значения по умолчанию для маршрута. И если вдруг в строке запроса мы не указали все параметры и попытались обратиться по адресу *http://localhost/*, то система маршрутизации вызовет метод Index контроллера Home, как указано в параметре defaults. Также, если мы не укажем метод контроллера, например, *http://localhost/Home/*, также будет вызван метод Index контроллера Home.

Поэтому если мы захотим, к примеру, чтобы у нас по умолчанию клиент обращался не к методу Index контроллера HomeController, а, например, к методу Show контроллера BookController, то мы можем соответственно изменить значения данного параметра:

|  |  |
| --- | --- |
| 1 | defaults: new { controller = "Book", action = "Show", id = UrlParameter.Optional } |

Последний параметр объявлен как необязательный id = UrlParameter.Optional, поэтому, если он не указан в строке запроса, он не будет учитываться и передаваться в словарь параметров RouteValueDictionary. Например, запрос *http://localhost/Home/Create/3* вызовет метод Create контроллера Home, передав в этот метод в качестве параметра число 3. В то же время запрос *http://localhost/Home/Create/* также вызовет метод Create контроллера Home, хотя последний параметр в нем не указан.

Таким образом, настройки по умолчанию позволяют нам не указывать в строке запроса полностью название контроллера и его метода. Но в случае если такие настройки не заданы, мы должны определять в строке запроса контроллер и его метод. Например, изменим установку маршрутов в файле **RouteConfig.cs** следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | public class RouteConfig      {          public static void RegisterRoutes(RouteCollection routes)          {              routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");                routes.MapHttpRoute(                  name: "DefaultApi",                  routeTemplate: "api/{controller}/{id}",                  defaults: new { id = RouteParameter.Optional }              );                routes.MapRoute(                  name: "Default",                  url: "{controller}/{action}"              );          }      } |

При запуске из Visual Studio или при запуске в браузере по адресу *http://mysyte.com/* мы получим информацию об ошибке. Ошибка будет состоять в том, что теперь нам полностью надо набирать в строке запроса адрес ресурса. Поэтому следующий адрес*http://mysyte.com/Home/Index* будет нормально работать (если у вас, конечно, определен контроллер Home с методом Index и соответствующим ему представлением).

И если мы теперь перейдем по адресу *http://localhost/Home/*, как мы это делали выше, то получим ошибку, так как у нас указан только одни сегмент. А в определении маршрута у нас указано два сегмента - *{controller}/{action}*. Если для параметров не определены значения по умолчанию, то строка запроса должна иметь такое же число сегментов, для которых не определены значения по умолчанию.

В то же время если запрос будет состоять из трех сегментов, например, *http://localhost/Home/Index/1*, то мы также получим ошибку, потому что число сегментов в запросе больше числа, определенного в шаблоне URL данного маршрута.

## Работа с маршрутами

### Создание новых маршрутов

Создать свой маршрут мы можем, например, как было уже показано, с помощью метода **MapRoute**. Например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class RouteConfig  {      public static void RegisterRoutes(RouteCollection routes)      {          //............................          routes.MapRoute(              name: "newRoute",              url: "{controller}/{action}"          );      }  } |

Либо мы можем создать объект Route и потом добавить его в коллекцию маршрутов RouteCollection. Определим два маршрута:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | public class RouteConfig  {      public static void RegisterRoutes(RouteCollection routes)      {          routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");            routes.MapHttpRoute(              name: "DefaultApi",              routeTemplate: "api/{controller}/{id}",              defaults: new { id = RouteParameter.Optional }          );            routes.MapRoute(              name: "Default",              url: "{controller}/{action}/{id}"          );            Route newRoute = new Route("{controller}/{action}", new MvcRouteHandler());          routes.Add(newRoute);      }  } |

Итак, здесь определены два маршрута. Первый - маршрут Default сопоставляется с запросами, имеющими три сегмента. Второй - newRouteсопоставляется с запросами, имеющими только два сегмента. В итоге вызов *http://localhost:5555/Home/Index/1* будет соответствовать первому маршруту, так как в нем определено три сегмента. А вызов *http://localhost:5555/Home/Index* - второму маршруту. Вызов*http://localhost:5555/Home* не будет соответствовать ни одному маршруту, так как у нас не определен маршрут, принимающий только один сегмент в шаблоне URL.

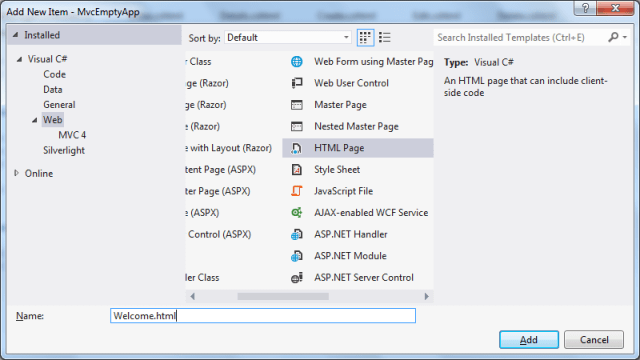
Правда, ситуация с этими двумя запросами в большей степени является искусственной, так как мы можем совместить их, просто определив параметр id как необязательный: routes.MapRoute(name: "Default", url: "{controller}/{action}/{id}", defaults: new { id = RouteParameter.Optional });.

При передаче значений по умолчанию для параметров мы должны учитывать позицию параметра. Движок маршрутизации использует значение по умолчанию только тогда, когда все последующие параметры также имеют значения по умолчанию. Так, если мы зададим следующий маршрут: routes.MapRoute(name: "Default", url: "{controller}/{action}/{id}", defaults: new { action = "Index" });, то запрос, содержащий два сегмента, например, *Home/2* не будет сопоставляться с данным маршрутом. Поэтому нам надо в данном случае указать значение по умолчанию также и для параметра id.

### Сопоставление запросов с файлами на диске

Ранее мы обращались к ресурсам сайта через маршруты. Но мы можем адресовать запросы также отдельным файлам сайта. Работа механизма маршрутизации такова, что сначала он смотрит, совпадает ли запрос с определенным файлом, хранящимся на сервере, и если такого файла не находит, тогда он начинает сопоставлять запрос с определенными маршрутами.

Итак, добавим в папку Content проекта новую html-страницу и назовем ее, например, *Welcome.html*



Определим какое-нибудь простенькое содержание страницы. Теперь мы можем обращаться к этой странице напрямую через запрос*Content/Welcome.html*.

#### Использование префиксов в строке запроса

Сегменты строки запроса необязательно должны нести только значения для параметров, определенных в маршруте. Мы также можем использовать различные префиксы в строке запроса и соответствующим образом настроить маршрут для обработки подобных запросов. Например, мы хотим, чтобы запрос содержал префикс Ru: *http://localhost:49326/Ru/Home/Index/1*. Тогда обрабатывающий этот запрос маршрут может выглядеть следующим образом:

|  |  |
| --- | --- |
| 1 | routes.MapRoute(name: "Default", url: "Ru/{controller}/{action}/{id}", defaults: new { id = RouteParameter.Optional } |

Кроме того, мы можем добавлять префиксы не в качестве отдельного сегмента, а к самому сегменту:

|  |  |
| --- | --- |
| 1 | routes.MapRoute(name: "Default", url: "Ru{controller}/{action}/{id}", defaults: new { id = RouteParameter.Optional } |

В этом случае строка запроса, соответствующая данному маршруту, может выглядеть так: *http://localhost:49326/RuHome/Index*

### Порядок определения новых маршрутов

При добавлении новых маршрутов важно учитывать их порядок. Обычно более специфические маршруты помещаются перед более общими. Например, предположим, мы определили следующие маршруты:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public static void RegisterRoutes(RouteCollection routes)  {      routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");        routes.MapRoute(name: "Default", url: "{controller}/{action}/{id}", defaults: new { id = RouteParameter.Optional });        routes.MapRoute( name: "Default2", url: "Ru{controller}/{action}/{id}", defaults: new { id = RouteParameter.Optional });  } |

Если мы отправим приложению запрос типа *http://localhost:49326/RuHome/Index* и если у нас в приложении не определен контроллер**RuHome**, то приложение вернет ошибку. Почему? Потому система маршрутизации пытается сопоставить запрос сначала с первым маршрутом. Если запрос не соответствует первому маршруту - тогда со вторым и так далее по списку маршрутов, пока не найдет нужный. В данном случае входящий запрос соответствует и первому и второму маршрутам. Однако первый маршрут будет искать контроллер по имени **RuHomeController** и, не найдя его, вернет ошибку. Поэтому чтобы подобная ситуация не произошла, надо сначала определить маршрут Default2, который более специфичен и не конфликтует с первым:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public static void RegisterRoutes(RouteCollection routes)  {      routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");      routes.MapRoute( name: "Default2", url: "Ru{controller}/{action}/{id}", defaults: new { id = RouteParameter.Optional });      routes.MapRoute(name: "Default", url: "{controller}/{action}/{id}", defaults: new { id = RouteParameter.Optional });  } |

Кроме того, мы можем использовать префиксы в качестве псевдонимов для контроллера и его метода:

|  |  |
| --- | --- |
| 1 | routes.MapRoute( name: "Default2", url: "Store/Buy", defaults: new { controller = "Home", action="Index" } ); |

Здесь для контроллера Home мы используем псевдоним Store, а для действия Index - псевдоним Buy. В итоге данный маршрут будет сопоставляться с таким запросом, как *Store/Buy*. А система маршрутизации будет обращаться по такому запросу к методу Index контроллера Home.

### Получение переданных параметров

Чтобы получить переданные значения для параметров маршрута, мы можем воспользоваться объектом **RouteData**. Например, если у нас определен стандартный маршрут routes.MapRoute(name: "Default", url: "{controller}/{action}/{id}", defaults: new { id = RouteParameter.Optional });, то мы можем получить значение параметра controller следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public string Index()  {      string controller = RouteData.Values["controller"].ToString();      return controller;  } |

Значения, передаваемые в качестве дополнительных значений, например, для параметра id, мы можем извлекать из самого параметра метода:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public ActionResult Index(int id)  {      ViewBag.OldId = id;      //или так      // ViewBag.OldId = RouteData.Values["id"];      return View();  } |

#### Передача различного числа параметров в запросе

Ранее мы ограничивались только тремя сегментами. Но если у нас скажем метод принимает два и более параметров:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public ActionResult Index(int id=1, string name="")  {      ViewBag.Name = name;      return View();  } |

То мы просто можем добавить в маршрут нужное нам количество параметров:

|  |  |
| --- | --- |
| 1 | routes.MapRoute(name: "Default", url: "{controller}/{action}/{id}/{name}", defaults: new { id = RouteParameter.Optional, name = RouteParameter.Optional }); |

Тогда данный маршрут будет обрабатывать запрос типа *Home/Index/1/name*, что эквивалентно следующей строке запроса: *Home/Index?id=1&name=name*

Кроме того, мы можем обозначить любое количество сегментов в запросе, чтобы не быть жестко привязанным к числу сегментов с помощью параметра **{\*catchall}**:

|  |  |
| --- | --- |
| 1 | routes.MapRoute(name: "Default", url: "{controller}/{action}/{id}/{\*catchall}",defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional}); |

Теперь мы можем обрабатывать запросы с любым количеством сегментов:

|  |  |
| --- | --- |
| **Запрос** | **Параметры запроса** |
| mysyte.com | controller=Home  action=Index |
| mysyte.com/Book | controller=Book  action=Index |
| mysyte.com/Book/Show | controller=Book  action=Show |
| mysyte.com/Book/Show/2 | controller=Book  action=Show  id=2 |
| mysyte.com/Book/Show/2/Oldedition | controller=Book  action=Show  id=2  catchall=Oldedition |
| mysyte.com/Book/Show/2/Oldedition/1960 | controller=Book  action=Show  id=2  catchall=Oldedition/1960 |

После получения значения для параметра catchall мы сами должны обработать его и получить уже значения для отдельных сегментов.

## Создание ограничений для маршрутов

Иногда возникает необходимость, более точно задать совпадение строки запроса для данного маршрута. Например, нам надо задать, чтобы имя контроллера обязательно начиналось с буквы "H". Тогда нам надо определить соответствующее ограничение для имени с помощью регулярных выражений:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | routes.MapRoute(      name: "Default",      url: "{controller}/{action}/{id}",      defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional },      constraints: new {controller="^H.\*" }  ); |

С помощью параметра constraints мы задаем ограничения маршрута. И если мы направим приложению запрос *Book/Index*, даже если у нас есть контроллер BookController с методом Index, то приложение вернет ошибку, так как имя контроллера попадает под ограничение, а другого маршрута, которому бы соответствовал запрос *Book/Index*, у нас не задано.

Подобным образом мы можем задать ограничения и для других параметров. Например, пусть параметр id состоит как минимум из двух цифр:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | routes.MapRoute(      name: "Default",      url: "{controller}/{action}/{id}/{\*catchall}",      defaults: new { controller = "Home", action = "Index" },      constraints: new { controller = "^H.\*", id = @"\d{2}" }  ); |

Теперь даже запрос *Home/Index/1* у нас не будет сопоставляться с маршрутом, так как параметр id состоит из одной цифры, а не из двух.

Также мы можем задать ограничение по типу метода. Например, мы хотим, чтобы обрабатывались только запросы для методов с типом GET:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | routes.MapRoute(      name: "Default",      url: "{controller}/{action}/{id}/{\*catchall}",      defaults: new { controller = "Home", action = "Index" },      constraints: new { controller = "^H.\*", id = @"\d{2}", , httpMethod=new HttpMethodConstraint("GET") }  ); |

Теперь будут обрабатываться только те запросы, которые представляют запросы типа GET.

**Создание собственных ограничений**

Чтобы создать собственное ограничение, нужно реализовать интерфейс IRouteConstraint с одним единственным методом Match, который имеет следующее определение:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public interface IRouteConstraint  {    bool Match(HttpContextBase httpContext, Route route, string parameterName,      RouteValueDictionary values, RouteDirection routeDirection);  } |

Ограничение маршрута применяет этот интерфейс IRouteConstraint, что вынуждает затем движок маршрутизации вызвать для ограничения маршрута метод IRouteConstraint.Match, чтобы определить, применяется ли данное ограничение к данному запросу или нет. Например, создадим ограничение, которое не будет пропускать запросу по некоторому url. Итак, добавим в приложение следующий класс:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public class CustomConstraint : IRouteConstraint  {      private string uri;      public CustomConstraint(string uri)      {          this.uri = uri;      }      public bool Match(HttpContextBase httpContext, Route route, string parameterName, RouteValueDictionary values, RouteDirection routeDirection)      {          return !(uri == httpContext.Request.Url.AbsolutePath);      }  } |

Здесь мы говорим, что если запрашиваемый ресурс совпадает со значением свойства httpContext.Request.Url.AbsolutePath, то запрос не будет сопоставляться с маршрутом. Тогда определение маршрута может выглядеть следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | routes.MapRoute(      name: "Default",      url: "{controller}/{action}/{id}/{\*catchall}",      defaults: new { controller = "Home", action = "Index" },      constraints: new { id = @"\d{2}", myConstraint = new CustomConstraint("/Home/Index/12") }  ); |

Теперь запрос */Home/Index/12* не будет обрабатываться, даже если он удовлетворяет всем остальным условиям и ограничениям.

**Игнорирование запросов**

По умолчанию в методе RegisterRoutes класса RouteConfig определена такая строка routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");. С помощью данного выражения мы запрещаем доступ к определенному ресурсу, находящемуся на сервере. Так, мы можем переписать предыдущий пример, где использовали ограничение для маршрута в виде запрета пути /Home/Index/12, следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public static void RegisterRoutes(RouteCollection routes)  {      routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");        routes.IgnoreRoute("Home/Index/12");        routes.MapRoute(          name: "Default",          url: "{controller}/{action}/{id}/{\*catchall}",          defaults: new { controller = "Home", action = "Index", id=RouteParameter.Optional } }      );  } |

Теперь также запрос *Home/Index/12* не будет обрабатываться и сопоставляться с определенным ресурсом на сервере.

## Генерация исходящих адресов URL

Еще одной стороной механизма маршрутизации является генерации исходящих адресов в приложении, например, при выводе их в представлении. Наиболее простой способ вывода адреса является использование якоря - элемента a:

|  |  |
| --- | --- |
| 1 | <a href="Home/Index/3" /> |

Но также мы можем использовать специальные хелперы рендеринга - **Html.ActionLink** и **Html.RouteLink**

**Html.ActionLink**

Хелпер ActionLink создает гиперссылку на действие контроллера. Если мы создаем ссылку на действие, определенное в том же контроллере, то можем просто указать имя действия:

|  |  |
| --- | --- |
| 1 | @Html.ActionLink("Жми здесь", "Show") |

Что создаст вам следующую разметку:

|  |  |
| --- | --- |
| 1 | <a href="/Home/Show">Жми здесь</a> |

Когда надо указать ссылку на действие из другого контроллера, то в хелпере ActionLink в качестве третьего аргумента имя контроллера. Например, ссылка на действие List контроллера Book будет создаваться так:

|  |  |
| --- | --- |
| 1 | @Html.ActionLink("Список книг", "List", "Book") |

Кроме того, если у нас в некотором методе Index контроллера Book определено несколько параметров:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class BookController : Controller  {      public string Index(string author="Толстой", int id=1)      {          return author + "  " + id.ToString();      }  } |

То перегруженная версия хелпера ActionLink позволяет передать параметр объекта (обычно анонимный тип) для параметра routeValues. Среда выполнения принимает свойства объекта и использует их для создания значений маршрутизации (имена свойств становятся именами параметров маршрута, а значения свойств представляют значения параметра маршрута). Создадим ссылку для вышеопределенного действия контроллера:

|  |  |
| --- | --- |
| 1  2  3 | @Html.ActionLink("Все книги", "Index", "Book", new { id=10}, null)  //или  @Html.ActionLink("Достоевский", "Index", "Book", new { author="Достоевский", id=5}, null) |

Последний параметр в данном хелпере является параметром htmlAttributes. Мы можем использовать этот параметр для установки значения атрибута элемента HTML. В данном случае передается значение null (то есть никаких атрибутов не устанавливается).

Теперь попробуем передать атрибуты, например, установить атрибуты id и class:

|  |  |
| --- | --- |
| 1 | @Html.ActionLink("Все книги", "Index", "Book", new {  author="Толстой",id = 10 }, new { id="Tolstoi", @class="link"}) |

Сгенерированная html-разметка будет выглядеть следующим образом:

|  |  |
| --- | --- |
| 1 | <a class="link" href="/Book/Index/10?author=%D0%A2%D0%BE%D0%BB%D1%81%D1%82%D0%BE%D0%B9" id="Tolstoi">Все книги</a> |

Обратите внимание на знак @ перед словом class: поскольку слово "class" является зарезервированным словом в C#, то для правильного рендеринга нам надо перед ним указать знак @.

**Html.RouteLink**

Хелпер RouteLink использует похожий шаблон, что и ActionLink: он принимает имя маршрута, но не требует аргументов для имени контроллера и имени действия. Так, первый пример с ActionLink эквивалентен следующему коду:

|  |  |
| --- | --- |
| 1 | @Html.RouteLink("Все книги", new { controller = "Book", action = "Index", author = "Толстой", id = 10 }, new { id = "Tolstoi", @class = "link" }) |

Чтобы использовать маршрут, нам просто надо указать имя определенного нами маршрута и затем определить при необходимости дополнительные параметры. Например, возьмем стандартный маршрут Default:

|  |  |
| --- | --- |
| 1  2  3  4  5 | routes.MapRoute(      name: "Default",      url: "{controller}/{action}/{id}",      defaults: new { controller = "Home", action = "Index", id=RouteParameter.Optional } }   ); |

Тогда создать ссылку мы можем, например, так:

|  |  |
| --- | --- |
| 1 | @Html.RouteLink("Все книги","Default",new { action = "Show" }) |

**URL-хелперы**

URL-хелперы похожи на хелперы ActionLink и RouteLink за тем исключением, что они не возвращают HTML, а создают пути URL и возвращают их в виде строк. Имеется три типа URL-хелперов:

* Action
* Content
* RouteUrl

Хелпер Action похож на ActionLink за тем исключением, что но не возвращает тег якоря. Например, следующий код отображает адрес URL, но не саму ссылку:

|  |  |
| --- | --- |
| 1 | @Url.Action("Index", "Book", new { author = "Толстой", id = 10 }, null) |

Хелпер RouteUrl использует тот же шаблон, что и Action, но как и RouteLink, принимает имя маршрута и аргументы для параметров маршрута:

|  |  |
| --- | --- |
| 1 | @Url.RouteUrl(new { controller = "Book", action = "Index", author = "Толстой", id = 10 }) |

Хелпер Content преобразует относительные пути в абсолютные. Пример использования хелпера Content можно увидеть в представлении \_Layout:

|  |  |
| --- | --- |
| 1  2 | <script src="@Url.Content("~/Scripts/jquery-1.5.1.min.js")"   type="text/javascript"></script> |

С помощью тильды (~) хелпер Content генерирует необходимый URL относительно расположения приложения. Без тильды URL мог бы стать некорректным, если бы вы перенесли приложение в другой виртуальный каталог.

## Области (Areas)

есмотря на то, что проект MVC по умолчанию представляет собой четкую структуру разделения на отдельные функциональные части - контроллеры, модели, представления, иногда для более удобной работы над приложением, особенно над большими приложениями, приложение делится на ряд областей (area).

Добавим в проект MVC область. Нажмем правой кнопкой мыши на проект и в появившемся меню выберем *Add->Area*. В окне добавления области дадим новой области название, например, Store. После этого в структуре проекта произойдет ряд изменений: в проект будет добавлена новая папка *Areas*, в которую в свою очередь будет добавлена папка Store - непосредственно для нашей новой области. Внутри папки Store фактически окажется мини-проект, в котором будут папки для контроллеров, моделей и представлений и класс регистрации области.

Откроем файл регистрации области *StoreAreaRegistration.cs*:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | using System.Web.Mvc;    namespace Routing.Areas.Store  {      public class StoreAreaRegistration : AreaRegistration      {          public override string AreaName          {              get              {                  return "Store";              }          }            public override void RegisterArea(AreaRegistrationContext context)          {              context.MapRoute(                  "Store\_default",                  "Store/{controller}/{action}/{id}",                  new { action = "Index", id = UrlParameter.Optional }              );          }      }  } |

Сгенерированные автоматически здесь определение маршрута показывает, как входящие запросы будут сопоставляться с контроллерами и действиями, определенными в данной области Store. Однако чтобы сопоставление запросов с областью происходило, также надо зарегистрировать все области в файле Global.asax.cs. Правда, вручную не придется это делать, так как при добавлении в проект первой области Visual Studio уже автоматически это делает. Вы можете открыть файл Global.asax.cs и увидеть изменения:

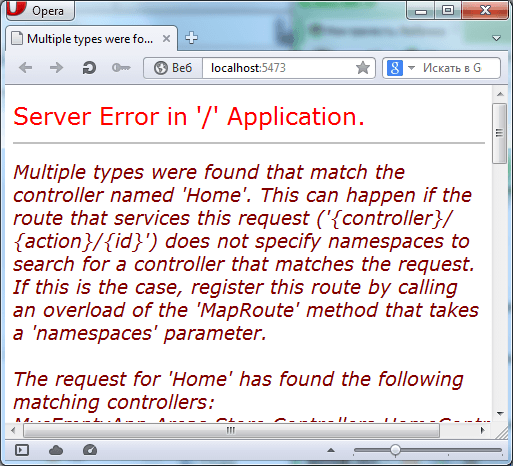
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | ............................  using System.Web.Routing;    namespace Routing  {      public class MvcApplication : System.Web.HttpApplication      {          protected void Application\_Start()          {              AreaRegistration.RegisterAllAreas();                FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);              RouteConfig.RegisterRoutes(RouteTable.Routes);              BundleConfig.RegisterBundles(BundleTable.Bundles);          }      }  } |

Чтобы протестировать нашу область, мы можем также добавить в папку Controllers, как и в основном проекте, новый контроллер и определить в нем действие. И также определить для действий контроллера представления. Допустим, у нас в области Store определен следующий контроллер AutoController:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class AutoController : Controller      {          public string Index()          {              return "AutoController";          }        } |

Тогда при запуске приложения мы можем обратиться к методу Index контроллера по адресу */Store/Auto/Index*, указав сначала имя области, а потом как обычно имя контроллера и его метода.

Однако что если мы захотим добавить в область контроллер Home с методом Index, как и в основном приложении. Поскольку у нас определен стандартный маршрут, который при запуске приложения будет отсылать нас к методу Index контроллера Home, то система маршрутизации окажется в двойственном положении: она не будет знать, к какому именно контроллеру обращаться - к тому, который определен в основном приложении, или к тому, который определен в области. В итоге вы увидите следующую картину:



Чтобы избежать подобной двойственности, в файле *RouteConfig.cs* в основном приложении надо непосредственно указать пространство имен контроллера Home, который будет вызываться при запуске приложения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public class RouteConfig      {          public static void RegisterRoutes(RouteCollection routes)          {              routes.MapRoute(                  name: "Default",                  url: "{controller}/{action}/{id}",                  defaults: new { controller="Home", action="Index", id = RouteParameter.Optional },                  namespaces: new[] { "Routing.Controllers" }              );          }      } |

В данном случае основное приложение находится в пространстве имен Routing, а контроллер Home - в пространстве именRouting.Controllers. Если бы мы хотели, чтобы у нас при запуске приложения отрабатывал метод Index, определенный в контроллере Home в области Store, то мы могли бы указать соответствующее пространство имен, которое в моем случае - Routing.Areas.Store.Controllers

**Генерация ссылок в областях**

При генерации ссылок в представлениях отдельных областей используются все те хелперы, например, Html.ActionLink, однако тут есть и некоторые особенности.

Чтобы сгенерировать ссылку на какое-либо действие контроллера, которые находятся внутри области, то мы указываем действие и контроллер (если действие находится в другом контроллере):

|  |  |
| --- | --- |
| 1 | @Html.ActionLink("Все книги", "Index", "Book", new { id=10}, null) |

В итоге будет сгенерирована ссылка: <a href="Store/Book/Index/10>Все книги</a>

Если же требуемое действие и контроллер находятся в другой области, то мы указываем область в параметре хелпера:

|  |  |
| --- | --- |
| 1 | @Html.ActionLink("Все книги", "List", new { area = "Library", controller="Book" }) |

Сгенерированная ссылка будет выглядеть так: <a href="Library/Book/List/>Все книги</a>

Если же метод и контроллер находятся в основном приложении, то для параметра area определяем пустую строку:

|  |  |
| --- | --- |
| 1 | @Html.ActionLink("Все книги", "Index", new { area = "", controller="Home" }) |

## Создание собственного обработчика маршрутов

Пред тем как приступить к созданию собственного обработчика маршрутов, посмотрим, что в целом представляет собой процесс маршрутизации.

Процесс маршрутизации состоит из следующих этапов:

1. Модуль UrlRoutingModule пытается сопоставить текущий запрос с маршрутами в таблице RouteTable.
2. Если сопоставление завершилось удачно, то модуль маршрутизации выбирает обработчик маршрутов сопоставленного маршрута - объект **IRouteHandler**.
3. Затем у объекта IRouteHandler вызывается метод GetHandler, который возвращает объект IHttpHandler, используемый для обработки запроса.
4. У обработчика IHttpHandler вызывается метод ProcessRequest для обработки запросов.

По умолчанию обработчик запросов или объект IRouteHandler представляет экземпляр класса MvcRouteHandler, который возвращает объектMvcHandler, применяющий интерфейс IHttpHandler. Этот объект MvcHandler отвечает за инициализацию контроллера, который потом уже вызывает одно из своих действий.

Однако мы можем переопределить это поведение, указав при определении маршрута свой обработчик маршрутов. Итак, сначала создадим сам обработчик. Для обработчика создадим какую-нибудь специальную папку в проекте и добавим в нее новый класс. Назовем егоMyRouteHandler:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | using System.Web;  using System.Web.Routing;    namespace Routing.RouteHandlers  {      public class MyRouteHandler : IRouteHandler      {          public IHttpHandler GetHttpHandler(RequestContext requestContext)          {              return new MyHttpHandler();          }      }        public class MyHttpHandler : IHttpHandler      {          public bool IsReusable          {              get { return false; }          }            public void ProcessRequest(HttpContext context)          {              context.Response.Write("Инопланетное послание : Привет мир!");          }      }  } |

Как было уже сказано про этапы маршрутизации, нам нужно собственно два класса: класс, реализующий интерфейс IHttpHandler, который и будет обрабатывать запрос; и класс, реализующий интерфейс IRouteHandler, который сопоставляется с маршрутом, и вызывает первый класс.

Теперь в классе RouteConfig пропишем маршрут, который будет обрабатываться нашим обработчиком, не забывая при этом импортировать пространство имен, в котором объявлен наш обработчик:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public class RouteConfig      {          public static void RegisterRoutes(RouteCollection routes)          {              routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");                routes.MapHttpRoute(                  name: "DefaultApi",                  routeTemplate: "api/{controller}/{id}",                  defaults: new { id = RouteParameter.Optional }              );                Route newRoute = new Route("{controller}/{action}", new MyRouteHandler());              routes.Add(newRoute);          }      } |

Теперь мы можем адресовать нашему приложению соответствующий запрос, например, *Home/Index* и браузер выведет нам нашу строку с посланием.

**Метаданные и валидация модели**

При работе с моделью в MVC большее значение играют метаданные. Они позволяют указать некоторую дополнительную информацию об объекте, например, о том, как отображать его свойства в представлении, или о том, как осуществлять валидацию вводимых данных.

## Аннотации данных для отображения свойств

Аннотации данных представляют собой атрибуты, которые вы можете найти в пространстве имен System.ComponentModel.DataAnnotations (хотя несколько атрибутов определено в других пространствах).

### Атрибут Display

Итак, предположим у нас имеется некоторая модель Book:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class Book  {      public virtual int Id { get; set; }      public virtual string Name { get; set; }      public virtual string Author { get; set; }      public virtual int Year { get; set; }  } |

И мы пытаемся получить из базы данных один объект этой модели:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public class HomeController : Controller  {      private BookContext db = new BookContext();      public ActionResult Index()      {          var firstBook = db.Books.ToList<Book>().First();          return View(firstBook);      }  } |

Используя соответствующий хелпер, мы выводим эту модель представлении: @Html.DisplayForModel():

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @model DataAnnotations.Models.Book    @{      ViewBag.Title = "Книга";  }    @Html.DisplayForModel() |

Обычный код, но тут мы сталкиваемся с проблемой:

Как видно, заголовки у свойств точно такие же, как и названия свойств, что не очень удобно. Ведь мы могли назвать свойства как угодно. К тому же мы не хотим, чтобы названия на русскоязычном сайте были на английском языке.

Конечно, в этом случае мы могли бы использовать @Html.LabelFor для настройки названия для каждого отдельного свойства, но в этом случае у нас бы увеличился в несколько раз объем кода, и мы не смогли бы одной строкой вывести всю информацию о модели. К тому же может потребоваться в нескольких представлениях выводить информацию о модели. Поэтому данный подход не оптимален.

И тут нам на помощь приходят аннотации данных, а именно атрибут **Display**. Нам достаточно поправить модель, указав у нее параметры отображения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | using System.ComponentModel.DataAnnotations;  ...........................................      public class Book      {          public virtual int Id { get; set; }          [Display(Name = "Название")]          public virtual string Name { get; set; }          [Display(Name = "Автор")]          public virtual string Author { get; set; }          [Display(Name = "Год")]          public virtual int Year { get; set; }      } |

Свойство Name атрибута Display содержит строку, которая будет отображаться вместо имени свойства. Больше нам ничего не надо менять. Запустим и увидим новые названия:

### Атрибут HiddenInput

В предыдущем примере у нас осталась одна проблемка - это поле Id. Иногда, конечно, может потребоваться вывод поля Id. Но, например, если бы мы выводили модель в режиме редактирования с помощью хелпера @Html.EditorForModel(), то данное поле было бы доступно для редактирования, что не очень хорошо, особенно когда идентификаторы не должны меняться. Чтобы скрыть это поле мы можем применить атрибут **HiddenInput**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | using System.ComponentModel.DataAnnotations;  using System.Web.Mvc;  ...........................................      public class Book      {         [HiddenInput (DisplayValue=false)]          public virtual int Id { get; set; }          [Display(Name = "Название")]          public virtual string Name { get; set; }          [Display(Name = "Автор")]          public virtual string Author { get; set; }          [Display(Name = "Год")]          public virtual int Year { get; set; }      } |

Свойство DisplayValue=false указывает, что надо скрыть данное поле. В итоге вы его не увидите:

При использовании хелперов редактирования (Html.EditorFor/Html.EditorForModel) для данного свойства будет сгенерировано скрытое поле:<input type="hidden" id="Id" name="Id" value="1" />

### Атрибут ScaffoldColumn

При редактировании модели атрибут HiddenInput полностью не скрывает поля, так как мы можем посмотреть исходный код страницы и найти соответствующие поля. Чтобы полностью скрыть свойство от хелперов, используется атрибут **ScaffoldColumn**:

|  |  |
| --- | --- |
| 1  2 | [ScaffoldColumn(false)]   public virtual int Id { get; set; } |

Теперь хелперы редактирования не увидят данное свойство и не создадут для него даже скрытое поле на странице.

### Атрибут DataType

Атрибут **DataType** позволяет предоставлять среде выполнения информацию об использовании свойства. Например, допустим, у нас есть свойство Password:

|  |  |
| --- | --- |
| 1  2 | [DataType(DataType.Password)]  public string Password { get; set; } |

Для свойства с атрибутом DataType.Password HTML-хелперы создают элемент ввода, у которого атрибут type имеет значение "password". Тогда в браузере вы при вводе данных вы не увидите вводимые символы:

Перечисление DataType может принимать несколько различных значений:

|  |  |
| --- | --- |
| **Значение** | **Описание** |
| Currency | Отображает текст в виде валюты |
| DateTime | Отображает дату и время |
| Date | Отображает только дату, без времени |
| Time | Отображает только время |
| Text | Отображает однострочный текст |
| MultilineText | Отображает многострочный текст (элемент textarea) |
| Password | Отображает символы с использованием маски |
| Url | Отображает строку URL |
| EmailAddress | Отображает электронный адрес |

### Атрибут UIHint

Данный атрибут указывает, какой будет использоваться шаблон отображения при создании разметки html для данного свойства. Шаблон управляет, как свойство будет рендерится на странице.

Имеются следующие встроенные шаблоны:

* **Boolean**

Хелперы редактирования создают флажок (checkbox) для булевых значений. Для значений типа bool? (nullable) создается элемент select с параметрами True, False и Not Set

Хелперы отображения генерируют те же элементы html, что и хелперы редактирования, только с атрибутом disabled

* **Collection**

Используется соответствующий шаблон для рендеринга каждого элемента коллекции. Причем элементы могут быть разных типов.

* **Decimal**

Хелперы редактирования создают однострочное текстовое поле - элемент input

* **EmailAddress**

Хелперы редактирования создают однострочное текстовое поле.

Хелперы отображения генерируют элемент ссылка, где атрибут href имеет значение mailto:url

* **HiddenInput**

Создается скрытое поле - элемент hidden input

* **Html**

Хелперы редактирования создают однострочное текстовое поле.

Хелперы отображения просто показывают текст

* **MultilineText**

Хелперы редактирования создают многострочное текстовое поле (элемент textarea)

* **Object**

Хелперы изучают свойства объекта и выбирают наиболее подходящие для него шаблоны.

* **Password**

Хелперы редактирования создают текстовое поле для ввода символов с использованием маски

Хелперы отображения показывают пароль как есть, без использования маски

* **String**

Хелперы редактирования создают однострочное текстовое поле

* **Url**

Хелперы редактирования создают текстовое поле

Хелперы отображения создают элемент ссылки для данного Url

Например, используем последний шаблон Url:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public class Book  {      [ScaffoldColumn(false)]      public virtual int Id { get; set; }        [Display(Name = "Название")]      [UIHint("Url")]      public virtual string Name { get; set; }        [Display(Name = "Автор")]      public virtual string Author { get; set; }        [Display(Name = "Год")]      public virtual int Year { get; set; }  } |

Тогда мы получим следующий результат:

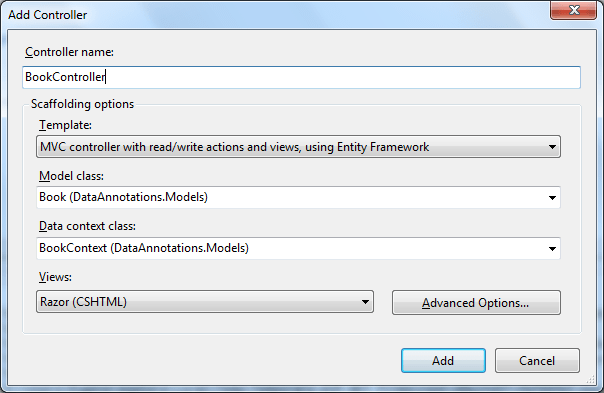
## Основы валидации

Итак, чтобы посмотреть, что из себя представляет валидация, создадим проект по шаблону Basic Application.

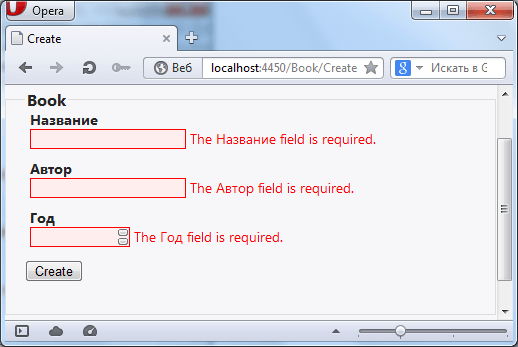
Добавим в наше приложение новую модель Book, которую частично мы разрабатывали ранее в этой главе:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public class Book  {      [ScaffoldColumn(false)]      public virtual int Id { get; set; }        [Required]      [Display(Name = "Название")]      public virtual string Name { get; set; }        [Required]      [Display(Name = "Автор")]      public virtual string Author { get; set; }        [Required]      [Display(Name = "Год")]      public virtual int Year { get; set; }  } |

Обратите внимание, что по сравнению с предыдущей версией данной модели, мы добавили атрибут **Required**. Теперь создадим контроллер, который будет управлять объектами данной модели. Сделаем контроллер (пусть контроллер называется допустим *BookController*) типизированным: в качестве шаблона выберем *MVC controller with read/write actions and views, using Entity Framework*, а в качестве класса модели укажем нашу модель Book:



В итоге у нас будет по умолчанию создан набор представлений по управлению объектами модели. Запустим приложение и обратимся к действию Create нашего контроллера (запрос *Book/Create*), которое должно добавлять новую запись в БД. Не заполняя поля, мы можем сразу нажать на кнопку отправки формы:



В данном случае мы задействовали механизм валидации, который используется в mvc по умолчанию. Каковы же основные моменты валидации в данном случае:

1. Использование атрибутов валидации при объявлении модели. Мы указали для каждого свойства атрибут Required, благодаря чему фреймворк знает, что данное свойство обязательно должно содержать некоторое значение.
2. Использование хелперов валидации. При каждом свойстве мы используем хелпер валидации Html.ValidationMessageFor:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <div class="editor-label">      @Html.LabelFor(model => model.Name)  </div>  <div class="editor-field">      @Html.EditorFor(model => model.Name)      @Html.ValidationMessageFor(model => model.Name)  </div> |

1. Благодаря чему и отображается сообщение об ошибке.
2. Чтобы настроить стиль отображения на стороне клиента, в файле Site.css указан стиль для соответствующих классов:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | .field-validation-error {      color: #f00;  }    .field-validation-valid {      display: none;  }    .input-validation-error {      border: 1px solid #f00;      background-color: #f00;  }    .validation-summary-errors {      font-weight: bold;      color: #f00;  }    .validation-summary-valid {      display: none;  } |

1. Валидация на стороне сервера. Эта была валидация на клиентской стороне. Однако если у вас отключен javascript а браузере, то вы не увидите буковок с ошибками, а форма благополучно отправится на сервер. Но во фреймворке предусмотрена также валидация на стороне сервера:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | [HttpPost]  public ActionResult Create(Book book)  {      if (ModelState.IsValid)      {          db.Books.Add(book);          db.SaveChanges();          return RedirectToAction("Index");      }      return View(book);  } |

1. С помощью свойства **ModelState.IsValid** мы узнаем, проходит модель валидацию или нет, и зависимости от результата совершаем те или иные действия.

Для просто демонстрации выбрано приложение по шаблону Basic, однако если вы работаете с проектом приложения по шаблону Empty, то вам придется добавлять все эти файлы jquery, код css и прочее, чтобы создать механизм валидации.

Таков в общем механизм валидации приложения по умолчанию. Теперь более подробно поговорим об основных моментах валидации. Начнем с атрибутов.

## Атрибуты валидации

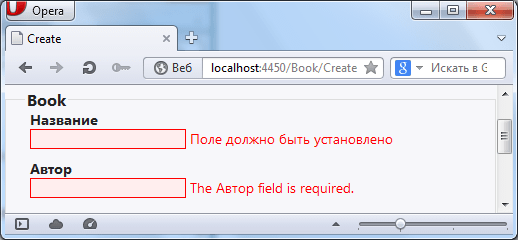
Рассмотрим атрибуты, участвующие в валидации модели на примере проекта приложения, созданного в прошлом параграфе данной главы.

### Атрибут Required

Применение этого атрибута к свойству модели означает, что данное свойство должно быть обязательно установлено.

Чтобы при валидации мы не получали несуразных сообщений об ошибке, этот атрибут позволяет настроить текст сообщения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public class Book  {      [ScaffoldColumn(false)]      public virtual int Id { get; set; }        [Required (ErrorMessage="Поле должно быть установлено")]      [Display(Name = "Название")]      public virtual string Name { get; set; }        [Required]      [Display(Name = "Автор")]      public virtual string Author { get; set; }        [Required]      [Display(Name = "Год")]      public virtual int Year { get; set; }  } |



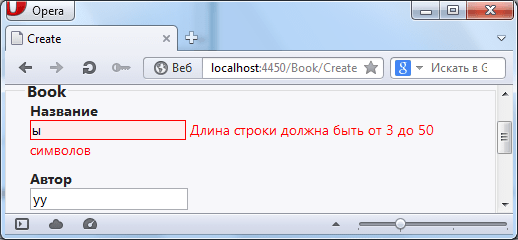
Таким образом, если мы явным образом не установим текст сообщения, то при выводе ошибки будет отображаться стандартный текст сообщения.

### Атрибут StringLength

Чтобы пользователь не мог ввести очень длинный текст, используется атрибут StringLength. Особенно это актуально, если в базе данных установлено ограничение на размер хранящихся строк.

Первым параметром идет максимальная допустимая длина строки. Именованные параметры, в частности MinimumLength и ErrorMessage, позволяют задать дополнительные опции отображения.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | public class Book  {      [ScaffoldColumn(false)]      public virtual int Id { get; set; }        [Required (ErrorMessage="Поле должно быть установлено")]      [StringLength (50, MinimumLength=3,ErrorMessage="Длина строки должна быть от 3 до 50 символов")]      [Display(Name = "Название")]      public virtual string Name { get; set; }        [Required]      [StringLength(50)]      [Display(Name = "Автор")]      public virtual string Author { get; set; }        [Required]      [Display(Name = "Год")]      public virtual int Year { get; set; }  } |



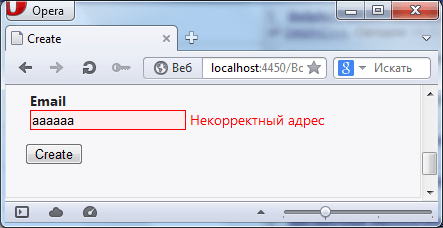
### Атрибут RegularExpression

Применение данного атрибута предполагает, что вводимое значение должно соответствовать указанному в этом атрибуте регулярному выражению.

Наиболее распространенный пример - это проверка корректности адреса электронной почты. Допустим, в некоторой модели у нас есть свойство Email:

|  |  |
| --- | --- |
| 1  2 | [RegularExpression(@"[A-Za-z0-9.\_%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}", ErrorMessage = "Некорректный адрес")]  public virtual string Email { get; set; } |

Если введенное значение не будет соответствовать регулярному выражению, то будет отображено сообщение об ошибке



### Атрибут Range

Атрибут Range определяет минимальные и максимальные ограничения для числовых данных.

|  |  |
| --- | --- |
| 1  2  3 | [Display(Name = "Год")]  [Range(1700,2000,ErrorMessage="Недопустимый год")]  public virtual int Year { get; set; } |

Атрибут Range может работать как с целочисленными значениями, так и с числами с плавающей точкой. А еще одна перегруженная версия его конструктора принимает параметр Type и две строки (которые позволяют создать диапазон возрастов).

|  |  |
| --- | --- |
| 1  2 | [Range(typeof(decimal), "0.00", "49.99")]  public decimal Price { get; set; } |

### Атрибут Remote

Атрибут Remote в отличие от предыдущих атрибутов находится в пространстве имен **System.Web.Mvc**. Он позволяет выполнять валидацию на стороне клиента с обратными вызовами на сервер.

Например, два пользователя не могут одновременно иметь одно и тоже значение UserName. Но с помощью валидации на стороне клиента трудно гарантировать, что введенное значение будет уникальным. А с помощью атрибута Remote мы можем послать значение свойстваUserName на сервер, а там оно уже сравнивается со значениями, находящимися в базе данных.

|  |  |
| --- | --- |
| 1  2 | [Remote("CheckUserName", "Account")]  public string UserName { get; set; } |

В атрибуте можно установить имя действия и имя контроллера, которые должны вызываться кодом на стороне клиента. Клиентский код посылает введенное пользователем значение для свойства UserName автоматически, а перегруженный конструктор атрибута позволяет указать дополнительные поля, значения которых надо посылать на сервер.

|  |  |
| --- | --- |
| 1  2  3  4  5 | public JsonResult CheckUserName(string username)  {      var result = Membership.FindUsersByName(username).Count == 0;      return Json(result, JsonRequestBehavior.AllowGet);  } |

Это действие контроллера принимает в качестве параметра имя свойства, подлежащего валидации, и возвращает true или false в форме объекта в формате JSON.

### Атрибут Compare

Атрибут Compare также находится в пространстве имен System.Web.Mvc. Он гарантирует, что два свойства объекта модели имеют одно и то же значение. Если, например, надо, чтобы пользователь ввел пароль дважды:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | [DataType(DataType.Password)]  public virtual string Password { get; set; }    [Compare("Password",ErrorMessage="Пароли не совпадают")]  [DataType(DataType.Password)]  public  virtual string PasswordConfirm { get; set; } |

Если пользователь введет второй раз другой пароль, отличный от первого, то он увидит ошибку:

## http://metanit.com/sharp/mvc/mvcpics/7.11.pngВалидация модели в контроллере

Кроме валидации на стороне клиента, мы можем осуществлять валидацию и внутри контроллера. Делается это с помощью проверки значения свойства **ModelState.IsValid**.

Объект ModelState сохраняет все значения, которые пользователь ввел для свойств модели, а также все ошибки, связанные с каждым свойством и с моделью в целом. Если в объекте ModelState имеются какие-нибудь ошибки, то свойство ModelState.IsValid возвратит false:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | [HttpPost]  public ActionResult Create(Book book)  {      if (ModelState.IsValid)      {          ViewBag.Message = "Valid";          db.Books.Add(book);          db.SaveChanges();          return RedirectToAction("Index");      }      ViewBag.Message = "Non Valid";      return View(book);  } |

Допустим, значение для свойства Name у нас не указано, и мы отсылаем значения модели на сервер. Поскольку к свойству применяется атрибут Required, то после привязки модели при валидации мы получим следующие значения:

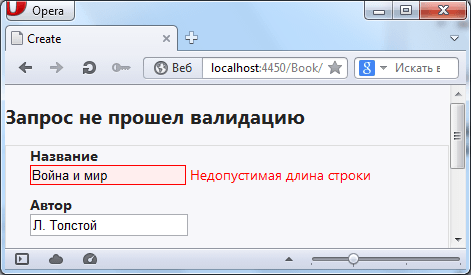
|  |  |
| --- | --- |
| 1  2  3 | ModelState.IsValid == false  ModelState.IsValidField("Name") == false  ModelState["Name"].Errors.Count > 0 |

Мы также можем проверять корректность значений отдельных свойств модели:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | [HttpPost]  public ActionResult Create(Book book)  {      if (string.IsNullOrEmpty(book.Name))      {          ModelState.AddModelError("Name", "Некорректное название книги");      }      else if (book.Name.Length > 5)      {          ModelState.AddModelError("Name", "Недопустимая длина строки");      }        if (ModelState.IsValid)      {          ViewBag.Message = "Валидация пройдена";          db.Books.Add(book);          db.SaveChanges();          return RedirectToAction("Index");      }        ViewBag.Message = "Запрос не прошел валидацию";      return View(book);  } |

В данном случае метод ModelState.AddModelError добавляет для свойства, указанного в качестве первого параметра (в данном случае Name) ошибку "Недопустимая длина строки". При использовании хелперов впоследствии мы можем вывести данное сообщение об ошибке:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <div class="editor-label">      @Html.LabelFor(model => model.Name)  </div>  <div class="editor-field">      @Html.EditorFor(model => model.Name)      @Html.ValidationMessageFor(model => model.Name)  </div> |



Кроме ошибок для конкретного свойства мы можем указывать ошибки на уровне модели. Выше мы видели, что у нас одно свойство имеет некорректное значение. Ошибки же на уровне модели связывают несколько свойств. Например,

|  |  |
| --- | --- |
| 1  2  3  4 | if (book.Author=="Л. Толстой" && book.Year > 1910)  {      ModelState.AddModelError("","Год не должен быть равен 1984");  } |

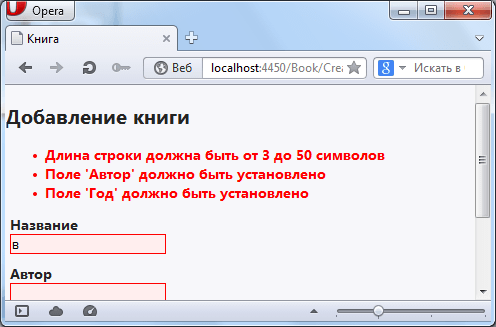
Оба свойства могут быть корректными в других ситуациях, но не вместе. Понятно, что Лев Толстой не мог написать, скажем, книги после своей смерти. Поэтому мы добавляем ошибку на уровне модели и в качестве первого параметра опускаем название свойства и указываем просто кавычки.

## Отображение ошибок валидации

Для отображения сообщений об ошибках при валидации модели мы можем использовать хелпер **Html.ValidationSummary**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | @using (Html.BeginForm()) {      @Html.ValidationSummary(true)            <div class="editor-label">              @Html.LabelFor(model => model.Name)          </div>          <div class="editor-field">              @Html.EditorFor(model => model.Name)              @Html.ValidationMessageFor(model => model.Name)          </div>            <div class="editor-label">              @Html.LabelFor(model => model.Author)          </div>          <div class="editor-field">              @Html.EditorFor(model => model.Author)              @Html.ValidationMessageFor(model => model.Author)          </div>            <div class="editor-label">              @Html.LabelFor(model => model.Year)          </div>          <div class="editor-field">              @Html.EditorFor(model => model.Year)              @Html.ValidationMessageFor(model => model.Year)          </div>            <p>              <input type="submit" value="Create" />          </p>    } |

В данном случае сообщения об ошибках выводятся над полями для ввода значений свойств:



Но данный хелпер имеет перегруженные версии, которые помогают настроить более точное отображение сообщений об ошибках:

|  |  |
| --- | --- |
| **Перегруженная версия** | **Описание** |
| Html.ValidationSummary() | Отображает общий список ошибок сверху |
| Html.ValidationSummary(bool) | Если параметр равен true, то вверху будут отображаться только сообщения об ошибках уровня модели, а специфические ошибки будут отображаться рядом с полями ввода. Если же параметр равен false, то вверху отображаются все ошибки. |
| Html.ValidationSummary(string) | Данная перегруженная версия хелпера отображает перед списком ошибок сообщение, которое передается в параметр string |
| Html.ValidationSummary(bool, string) | Сочетает две предыдущие перегруженные версии |

Также мы можем использовать хелпер Html.ValidationMessageFor для вывода сообщения об ошибке для отдельного свойства, причем рядом с полем для ввода свойства, а не в сводке ошибок сверху:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | @Html.ValidationSummary(true)    <div class="editor-label">      @Html.LabelFor(model => model.Name)  </div>  <div class="editor-field">      @Html.EditorFor(model => model.Name)      @Html.ValidationMessageFor(model => model.Name)  </div>    <div class="editor-label">       @Html.LabelFor(model => model.Author)  </div>  <div class="editor-field">      @Html.EditorFor(model => model.Author)      @Html.ValidationMessageFor(model => model.Author)  </div> |

### Стилизация ошибок

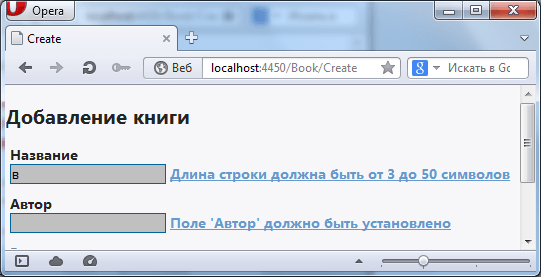
Еще один важный момент отображения ошибок - это их стилизация. То, что мы видим ошибки в красном цвете и границы полей ввода также в красном цвете, не жестко установлено, и мы все это можем изменить. В файле стилей **Site.css** в проектах, созданных по шаблону Basic и Internet Application, мы можем найти секцию стилей, которая как раз и отвечает за стилизацию (при использования проекта по типу Empty нам самим придется добавить стили):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | /\* Styles for validation helpers  -----------------------------------------------------------\*/  .field-validation-error {      color: #f00;  }    .field-validation-valid {      display: none;  }    .input-validation-error {      border: 1px solid #f00;      background-color: #fee;  }    .validation-summary-errors {      font-weight: bold;      color: #f00;  }    .validation-summary-valid {      display: none;  } |

Это классы используются полями, генерируемыми хелперами валидации. И изменяя данные стили, мы можем управлять отображением ошибок. Итак, изменим два стиля - стиль класса .field-validation-error (он управляет отображением ошибок рядом с текстовыми полями) и стиль класса стиль класса .input-validation-error, который управляет отображением поля ввода в режиме ошибки:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | .field-validation-error {      color: #6699CC;      font-weight:bold;      text-decoration:underline;  }    .input-validation-error {      border: 1px solid #006699;      background-color: silver;  } |

В итоге при возникновении ошибок валидации мы увидим следующую картину:



## Создание собственной логики валидации

Хотя встроенные атрибуты валидации охватывают приличное количество ситуаций, которые могут возникнуть, но все же их бывает недостаточно. Но mvc фреймворк настолько гибкий, что позволяет создавать собственные атрибуты валидации и закладывать в них свою логику.

Все атрибуты валидации образованы от базового класса ValidationAttribute, который находится в пространстве именSystem.ComponentModel.DataAnnotations. Поэтому именно от этого класса мы будем образовывать свой атрибут.

Допустим, нам надо, чтобы какая-нибудь книга была написана ограниченным кругом авторов. Создадим соответствующий атрибут, который будет это проверять:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | using System.ComponentModel.DataAnnotations;    namespace Mvc4BasicApplication.Annotations  {      public class MyAuthorsAttribute : ValidationAttribute      {          //массив для хранения допустимых авторов          private static string[] myAuthors;            public MyAuthorsAttribute(string[] Authors)          {              myAuthors = Authors;          }            public override bool IsValid(object value)          {              if (value != null)              {                  string strval = value.ToString();                  for (int i = 0; i < myAuthors.Length; i++)                  {                      if (strval == myAuthors[i])                          return true;                  }              }              return false;          }      }  } |

Чтобы применить логику валидации, надо переопределить метод IsValid, предоставленный базовым классом.

Логика атрибута довольно проста - мы получаем массив допустимых авторов. А при получении значения валидируемого свойства проверяем, имеется ли оно в этом массиве авторов, в зависимости от чего возвращаем true или false.

Использование атрибута аналогично использованию других атрибутов валидации:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | public class Book  {      [ScaffoldColumn(false)]      public virtual int Id { get; set; }        [Required (ErrorMessage="Поле должно быть установлено")]      [Display(Name = "Название")]      public virtual string Name { get; set; }        [Required]      [MyAuthors(new string[] { "Л. Толстой", "А. Пушкин", "Ф. Достоевский", "И. Тургенев" }, ErrorMessage="Недопустимый автор")]      [Display(Name = "Автор")]      public virtual string Author { get; set; }        [Required]      [Display(Name = "Год")]      [Range(1700,2000,ErrorMessage="Недопустимый год")]      public virtual int Year { get; set; }  } |

В итоге свойство будет невалидно, если был введен любой автор, не входящий в список.

### Атрибуты валидации на уровне модели

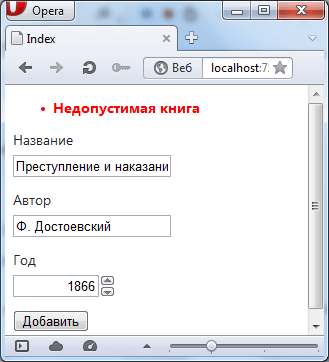
Атрибуты валидации на уровне модели применяются к проверке комбинации свойств. Например, мы не хотим, чтобы пользователь мог ввести определенную книгу:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public class NotAllowedAttribute : ValidationAttribute  {      public override bool IsValid(object value)      {          Book b = value as Book;          if (b.Name=="Преступление и наказание" && b.Author=="Ф. Достоевский" && b.Year==1866)          {              return false;          }          return true;      }  } |

Тогда устанавливаем атрибут для всей модели:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | [NotAllowedAttribute(ErrorMessage="Недопустимая книга")]  public class Book  {      [ScaffoldColumn(false)]      public virtual int Id { get; set; }        [Required (ErrorMessage="Поле должно быть установлено")]      [Display(Name = "Название")]      public virtual string Name { get; set; }      //.......................  } |

При попытке ввести указанные данные для книги мы получим ошибку:



### Самовалидация и IValidatableObject

Самовалидация представляет собой процесс, при котором модель запускает механизм валидации из себя самой. И сама инкапсулирует всю логику валидации.

Для этого класс модели должен реализовать интерфейс **IValidatableObject**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | public class Book : IValidatableObject  {      [ScaffoldColumn(false)]      public virtual int Id { get; set; }        [Display(Name = "Название")]      public virtual string Name { get; set; }        [Display(Name = "Автор")]      public virtual string Author { get; set; }        [Display(Name = "Год")]      public virtual int Year { get; set; }        public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)      {          List<ValidationResult> errors = new List<ValidationResult>();            if (string.IsNullOrEmpty(this.Name))          {              errors.Add(new ValidationResult("Введите название книги"));          }          if (string.IsNullOrEmpty(this.Author))          {              errors.Add(new ValidationResult("Введите автора книги"));          }          if (this.Year == null || this.Year<1700 || this.Year>2000)          {              errors.Add(new ValidationResult("Недопустимый год"));          }            return errors;      }  } |

В данном случае нам надо реализовать метод Validate и возвратить коллекцию объектов ValidationResult, которые и будут содержать все ошибки валидации.

### Создание своего провайдера валидации

В дополнение к выше рассмотренным методам переопределения механизма валидации мы также можем создать свой провайдер валидации. Для этого мы должны создать класс производный от класса ModelValidatorProvider и переопределить его метод GetValidators.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public class MyValidationProvider : ModelValidatorProvider  {      public override IEnumerable<ModelValidator> GetValidators(ModelMetadata metadata, ControllerContext context)      {          if (metadata.ContainerType == typeof(Book))          {              return new ModelValidator[] { new BookPropertyValidator(metadata,context)};          }            if (metadata.ModelType == typeof(Book))          {              return new ModelValidator[] { new BookValidator(metadata, context) };          }            return Enumerable.Empty<ModelValidator>();      }  } |

Классы BookPropertyValidator и BookValidator мы расмотрим чуть позже, а пока посмотрим общую логику класса.

Метод GetValidators вызывается для каждого свойства и отдельно для всей модели. Поэтому мы используем два валидатора -BookPropertyValidator (для свойств модели) и BookValidator (для модели в целом).

С помощью переданного в качестве параметра объекта ModelMetadata мы получаем некоторую информацию касательно объектов валидации. Так, мы можем получить доступ к следующим свойствам данного объекта:

* **CotainerType**. Это свойство возвращает тип валидируемой модели, которая содержит указанное свойство.
* **PropertyName**. Это свойство возвращает имя валидируемого свойства
* **ModelType**. Это свойство возвращает тип объекта модели

Теперь посмотрим на класс BookPropertyValidator, который инкапсулирует логику валидации для отдельных свойств:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | public class BookPropertyValidator : ModelValidator  {      public BookPropertyValidator(ModelMetadata metadata, ControllerContext context)          : base(metadata, context)      { }        public override IEnumerable<ModelValidationResult> Validate(object container)      {          Book b = container as Book;          if (b != null)          {              switch (Metadata.PropertyName)              {                  case "Name" :                      if (string.IsNullOrEmpty(b.Name))                      {                          return new ModelValidationResult[]{                              new ModelValidationResult { MemberName="Name", Message="Введите название книги"}                          };                      }                      break;                  case "Author":                      if (string.IsNullOrEmpty(b.Author))                      {                          return new ModelValidationResult[]{                              new ModelValidationResult { MemberName="Author", Message="Введите автора книги"}                          };                      }                      break;                  case "Year":                      if (b.Year>2000 || b.Year<1700)                      {                          return new ModelValidationResult[]{                              new ModelValidationResult { MemberName="Year", Message="Недопустимый год"}                          };                      }                      break;              }          }          return Enumerable.Empty<ModelValidationResult>();      }  } |

В методе Validate мы определяем валидируемые свойства и вызываем соответствующие действия по валидации объекта. Затем в объектModelValidationResult добавляем сведения касательно возникшей ошибки: свойство MemberName указывает на имя валидируемого свойства, а свойство Message - на сообщение об ошибке валидации.

Похожим образом выглядит валидатор для всей модели - BookValidator, только в объект ModelValidationResult в качестве значения свойстваMemberName мы передаем пустые кавычки:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | public class BookValidator : ModelValidator  {      public BookValidator(ModelMetadata metadata, ControllerContext context)          : base(metadata, context)      { }        public override IEnumerable<ModelValidationResult> Validate(object container)      {          Book b = (Book)Metadata.Model;            List<ModelValidationResult> errors = new List<ModelValidationResult>();            if (b.Name == "Преступление и наказание" && b.Author == "Ф. Достоевский" && b.Year == 1866)          {              errors.Add(new ModelValidationResult { MemberName = "", Message = "Недопустимая книга" });          }          return errors;      }  } |

Последним шагом будет регистрация своего провайдера валидации в файле Global.asax:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public class MvcApplication : System.Web.HttpApplication  {      protected void Application\_Start()      {          ModelValidatorProviders.Providers.Add(new MyValidationProvider());            AreaRegistration.RegisterAllAreas();            Database.SetInitializer(new BookDbInitializer());              WebApiConfig.Register(GlobalConfiguration.Configuration);          FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);          RouteConfig.RegisterRoutes(RouteTable.Routes);          BundleConfig.RegisterBundles(BundleTable.Bundles);      }  } |

Теперь мы можем освободить нашу модель от всех атрибутов, так как вся логика валидации у нас уже заложена в провайдере:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public class Book  {      [ScaffoldColumn(false)]      public virtual int Id { get; set; }        [Display(Name = "Название")]      public virtual string Name { get; set; }        [Display(Name = "Автор")]      public virtual string Author { get; set; }        [Display(Name = "Год")]      public virtual int Year { get; set; }  } |

# Фильтры

## Фильтры. Введение

Иногда возникает необходимость внести в приложение некоторую логику, которая должна отрабатывать до вызова действий контроллера. В частности, например, пользователь вызывает определенное действие некоторого контроллера, и нам необходимо проверять, авторизовался ли данный пользователь, и потом уже выполнять определенные действия. Для этого и применяются фильтры.

Фильтры реализованы как атрибуты, благодаря чему позволяют уменьшить объем кода в контроллере. Данные атрибуты могут применяться как ко всему классу, так и к отдельным его методам, свойствам и полям. В качестве примера вы можете создать новый проект приложения по шаблону Internet Application - в этом случае в папке Contollers у вас по умолчанию будет создан контроллер **AccountController**:

|  |  |
| --- | --- |
| 1  2  3  4  5 | [Authorize]   public class AccountController : Controller   {           ................................   } |

В данном случае **атрибут [Authorize]** как раз и будет представлять фильтр авторизации. Если бы мы не применили данный фильтр, то нам бы пришлось вручную писать логику проверки, авторизован ли пользователь или нет.

### Применение фильтров

Как вы видели из примера выше, для применения фильтра, надо атрибут поставить над классом контроллера - в этом случае данный фильтр будет применяться ко всем методам данного контроллера.

Но, как уже выше говорилось, мы можем применить фильтр на уровне отдельного метода:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public class HomeController : Controller  {      [Authorize]      public ActionResult Index()      {          ViewBag.Message = "Modify this template to kick-start your ASP.NET MVC application.";            return View();      }  } |

### Базовые типы фильтров

В MVC реализовано четыре типа фильтров:

|  |  |  |  |
| --- | --- | --- | --- |
| **Тип фильтров** | **Реализуемый интерфейс** | **Стандартная реализация** | **Описание** |
| Фильтры авторизации | IAuthorizationFilter | AuthorizationAttribute | Фильтр, определяющий, аутентифицирован ли клиент и имеет ли он доступ к данному ресурсу. Данный фильтр запускается до выполнения любого другого фильтра или метода действий |
| Фильтры действий | IActionFilter | ActionFilterAttribute | Фильтр, применяемый к действиям. Может запускаеться как до, так и после выполнения метода действий |
| Фильтры результатов действий | IResultFilter | ResultFilterAttribute | Фильтр, применяемый к результатам действий. Может запускаться как до, так и после выполнения результата действия |
| Фильтры исключений | IExceptionFilter | HandleErrorAttribute | Атрибут для обработки исключений, выбрасываемых методом действий и результатом действий |

При вызове метода действий фреймворк MVC сначала смотрит, применяются ли к данному действию какие-либо атрибуты, и если применяются, то в начале отрабатывают данные атрибуты.

Теперь разберем первый фильтр - фильтр авторизации.

## Фильтр авторизации

Фильтры авторизации отрабатывают до запуска остальных фильтров и вызова методов действий. Они реализуют интерфейсIAuthorizationFilter:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | using System.Web.Mvc  {      public interface IAuthorizationFilter      {          void OnAuthorization(AuthorizationContext filterContext);      }  } |

И если при получении запроса окажется, что к запрашиваемому действию контроллера применяется данный фильтр, то сначала срабатывает метод OnAuthorization данного интерфейса. И если фильтр одобрит запрос, то далее вызывается действие. Иначе действие не будет работать.

Для понимания работы фильтров авторизации рассмотрим стандартное приложение MVC по шаблону Internet Application, которое по умолчанию содержит всю логику авторизации и применение этого фильтра. В рамках данного раздела мы не будем подробно останавливаться на механизмах авторизации и аутентификации, лишь рассмотрим их в той мере, в какой они касаются фильтров авторизации.

Во-первых, откроем файл Web.config. Где-то в районе 22-23 строки вы увидите настройки аутентификации (Часто путают такие понятия как аутентификация и авторизация. Аутентификация представляет верификацию пользователя с помощью механизма ввода логина (имени пользователя/пароля и т.д.), а авторизация - это верификация тех действий, которые пользователь может совершать в отношении сайта. Обычно с авторизацией связывают систему разделения ролей.):

|  |  |
| --- | --- |
| 1  2  3 | <authentication mode="Forms">     <forms loginUrl="~/Account/Login" timeout="2880" />   </authentication> |

В данном случае указывается, что будет использоваться **аутентификация форм**. А в качестве пути, по которому можно произвести авторизацию на сайте, будет использоваться путь */Account/Login* - то есть авторизовать вас или нет будет решать метод Login контроллера Account.

Атрибут **timeout** указывает время в минутах, которое пользователь может быть авторизован на сайте после успешной аутентификации.

Кроме аутентификации форм существует аутентификация Windows, однако ее можно использовать преимущественно в приложениях по шаблону Intranet Application, где для идентификации пользователей используются учетные записи Windows. Однако к Internet Application аутентификация Windows не применяется.

Также для аутентификации и авторизации нам необходима модель, с которой бы сопоставлялись учетные записи пользователей. В папке models вы можете увидеть файл AccountModels.cs, который по умолчанию содержит уже три модели - для регистрации (**RegisterModel**), логина (**LoginModel**) и смены пароля (**ChangePasswordModel**).

Весь процесс аутентификации выполняется контроллером **AccountController**, к которому применяется фильтр авторизации. Рассмотрим действие **Login**, которое и предназначено для аутентификации пользователя:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | [AllowAnonymous]  public ActionResult Login(string returnUrl)  {      ViewBag.ReturnUrl = returnUrl;      return View();  }    //  // POST: /Account/Login    [HttpPost]  [AllowAnonymous]  [ValidateAntiForgeryToken]  public ActionResult Login(LoginModel model, string returnUrl)  {      if (ModelState.IsValid && WebSecurity.Login(model.UserName, model.Password, persistCookie: model.RememberMe))      {                  return RedirectToLocal(returnUrl);      }        // If we got this far, something failed, redisplay form      ModelState.AddModelError("", "The user name or password provided is incorrect.");      return View(model);  } |

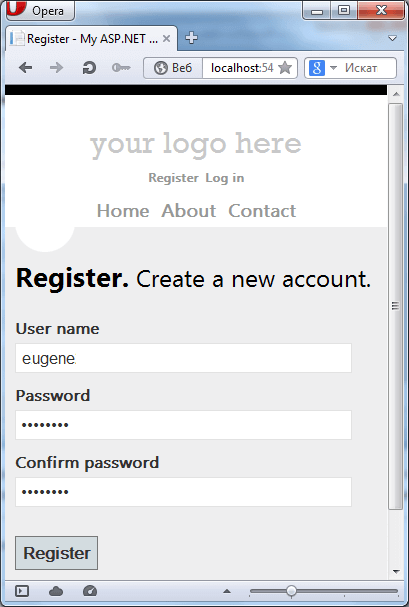
### Атрибут AllowAnonymous

Во-первых, обратите внимание на атрибут **AllowAnonymous**, который установлен для большинства методов контроллера. Поскольку у нас к контроллеру применяется атрибут Authorize, то фреймворк сначала будет смотреть, есть ли у пользователя необходимые права для доступа к методу контроллера. Однако в этом случае мы попадаем в замкнутый круг - чтобы авторизоваться, надо обратиться к методу Login, но чтобы обратиться к этому методу, уже надо быть авторизованным. Поэтому применяется атрибут AllowAnonymous, который открывает публичный доступ к методу контроллера.

### Аутентификация

Перейдем к POST-методу, где мы получаем введенные данные логина и пароля: первая часть конструкции if (ModelState.IsValidосуществляет валидацию полученной модели. А вторая часть конструкции - метод WebSecurity.Login(model.UserName, model.Password, persistCookie: model.RememberMe) получает логин и пароль пользователя и сравнивает их с теми пользователями, которые уже имеются. И если пользователь с такими данными существует, то пользователь аутентифицируется, и устанавливаются куки.

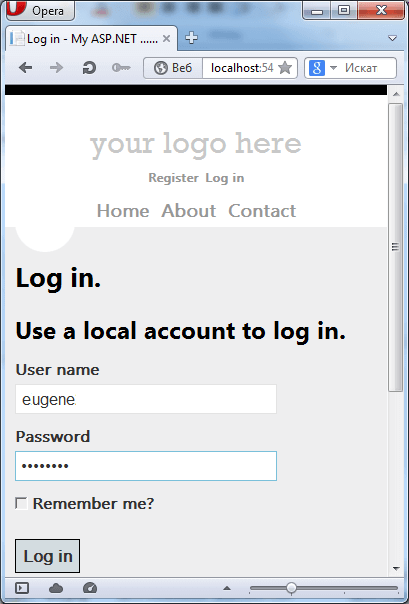
Если вы запустите приложение, вы можете зарегистрировать нового пользователя. И потом уже от его имени осуществлять вход на сайт.



Однако у нас пока по сути фильтр авторизации не применяется ни к чему, кроме контроллера AccountController. Поэтому откроем контроллер HomeController и к методу Index применим атрибут Authorize:

|  |  |
| --- | --- |
| 1  2  3  4  5 | [Authorize]  public ActionResult Index()  {      .............  } |

В результате после запуска приложения мы попадем не на представление Index.cshtml, а на представление Login.cshtml. Так как фреймворк видит, что мы не аутентифицированы , и поэтому будет перенаправлять нас на метод Login контроллера Account, как определено в файле конфигурации проекта.



Лишь после удачного входа на сайт нам станет доступен ресурс Home/Index.

### Взаимодействие с БД

Теперь еще один интересный вопрос - где это все сохраняется? Если вы посмотрите на файл конфигурации, то можете увидеть в секции**connectionStrings** настройки всех возможных подключений проекта к базам данных. По умолчанию создается подключение**DefaultConnection**. В моем случае оно выглядит так для проекта с названием *TestMvcApplication*, создаваемого в Visual Studio 2012:

|  |  |
| --- | --- |
| 1  2  3  4 | <connectionStrings>      <add name="DefaultConnection" connectionString="Data Source=(LocalDb)\v11.0;Initial Catalog=aspnet-TestMvcApplication-20130423121813;Integrated Security=SSPI;              AttachDBFilename=|DataDirectory|\aspnet-TestMvcApplication-20130423121813.mdf" providerName="System.Data.SqlClient" />    </connectionStrings> |

Опреlеление строки подключения для Visual Studio 2010 выглядело бы примерно так:

|  |  |
| --- | --- |
| 1  2  3 | <connectionStrings>      <add name="DefaultConnection" providerName="System.Data.SqlClient" connectionString="Data Source=.\SQLEXPRESS;Initial Catalog=aspnet-Mvc4InternetApplication-20120828133008;Integrated Security=SSPI" />    </connectionStrings> |

В Visual Studio 2012 при использовании шаблона Internet Application, содержащего инфраструктуру для работы с учетными записями, используется подход CodeFirst - то есть при первом обращении к моделям создается база данных в папке App\_Data. Мы можем просмотреть БД, перейдя в окно Database Explorer (либо визуально мы модем увидеть ее в проводнике).

По имеющимся моделям в базе создается несколько таблиц, которые предназначены для хранения учетных данных, как пользователи, роли и т.д.

Таблица **UserProfile** содержит список пользователей, зарегистрированных для данного приложения.

Учетные данные для каждого пользователя, как например, пароль и другие, хранятся в таблице **webpages\_Membership**, а таблица**webpages\_Roles** содержит роли, определенные в системе. По умолчанию таблица не содержит каких-либо ролей, поэтому мы можем добавить их:

Затем поскольку непонятно, какой пользователь какую роль в системе выполняет, нам надо связать роли и пользователей. Для этого откроем таблицу **webpages\_UsersInRoles** и сопоставим id пользователей с id ролей:

После этого мы можем использовать роли в приложении.

(В Visual Studio 2010 для определения и назначения ролей можно воспользоваться средством конфигурации ASP.NET Configuration, которое можно запустить, выбрав в меню Visual Studio пункт Project->ASP.NET Configuration)

### Использование ролей при авторизации

Авторизация в mvc обладает большой гибкостью, так как мы можем настроить доступ к ресурсам сайта для отдельных пользователей или ролей. Для этого мы можем использовать два свойства атрибута AuthorizeAtribute:

* **Users** - содержит перечисление имен пользователей, которым разрешен вход
* **Roles** - содержит перечисление имен ролей, которым разрешен вход

Использование свойств Users и Roles:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | [Authorize (Users="eugene, sergey")]  public ActionResult Index()  {      .............  }  [Authorize (Roles="admin")]  public ActionResult Create()  {      .............  }  [Authorize (Roles="admin, moderator", Users="eugene, sergey")]  public ActionResult Edit()  {      .............  } |

В случае если роли или пользователи не указаны, то доступ к методам контроллера имеет любой аутентифицированный пользователь.

### Создание собственного фильтра авторизации

Выше мы посмотрели, как работает фильтр авторизации. Однако за кадром остались принципы работы данного фильтра. Для этого мы создадим свой фильтр.

В реальных приложениях не рекомендуется писать свой код или классы, которые призваны обеспечивать безопасность. Либо это должны делать высококвалифицированные разработчики. Поскольку нередко большинство взломов приходят от того, что программисты с малым опытом в данных вопросах создают не самый качественный код. Поэтому лучше полагаться на проверенные и протестированные стандартные реализации фреймворка MVC.

Наиболее простой и безопасный способ создания своего фильтра авторизации - это наследование класса от атрибута AuthorizeAtribute и переопределение метода AuthorizeCore:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | using System;  using System.Web;  using System.Web.Mvc;    namespace Mvc4InternetApplication.Filters  {      public class MyAuthorizedAttribute : AuthorizeAttribute      {          private string[] allowedUsers;          private string[] allowedRoles;            public MyAuthorizeAttribute(string[] users, string[] roles)          {              allowedUsers = users;              allowedRoles = roles;          }            protected override bool AuthorizeCore(HttpContextBase httpContext)          {              return httpContext.Request.IsAuthenticated &&                  allowedUsers.Contains(httpContext.User.Identity.Name) &&                  Role(httpContext);          }            private bool Role(HttpContextBase httpContext)          {              if (allowedRoles.Length > 0)              {                  for (int i = 0; i < allowedRoles.Length; i++)                  {                      if (httpContext.User.IsInRole(allowedRoles[i]))                          return true;                  }                  return false;              }              return true;          }      }  } |

Массивы пользователей и ролей помогают провести детальную аутентификацию и авторизацию. (Хотя, как вы видели, у атрибута уже есть свойства Users и Roles.) С помощью объекта контекста HttpContextBase мы получаем все данные запроса и ответа, а также устанавливаем, является ли запрос аутентифицированным. Применение фильтра подобно применению стандартной версии:

|  |  |
| --- | --- |
| 1  2  3  4  5 | [MyAuthorize (Roles="admin, moderator", Users="eugene, sergey")]  public ActionResult Edit()  {      .............  } |

Переопределение атрибута AuthorizeAtribute наиболее безопасный способ создания механизма авторизации, по сравнению, скажем, с реализацией в своем классе интерфейса IAuthorizationFilter. Поэтому в случаях, когда вам надо создать свой класс атрибута авторизации, рекомендуется именно этот способ.

Хотя в предыдущем примере переопределение по сути не имело смысла, однако все-таки может потребоваться реализация своего атрибута. Например, чтобы администраторам сервера сайта не приходилось каждый раз набирать пароль при входе на сайт, мы можем сделать следующую реализацию атрибута:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class LocalAuthorizeAttribute : AuthorizeAttribute  {      protected override bool AuthorizeCore(HttpContextBase httpContext)      {          return httpContext.Request.IsLocal || base.AuthorizeCore(httpContext);      }  } |

И также использовать роли и пользователей в приложении:

|  |  |
| --- | --- |
| 1  2  3  4  5 | [LocalAuthorize (Roles="admin", Users="eugene, sergey")]  public ActionResult Edit()  {      .............  } |

Однако в этом случае заходить на сайт смогут не только указанные роли и пользователи, но и локальные пользователи для данной машины (что мы узнаем с помощью свойства **httpContext.Request.IsLocal**).

## Фильтры исключений

Фильтры исключений срабатывают, если при выполнении метода действия будет выброшено необработанное исключение.

С одной стороны, мы могли поместить всю логику выполнения метода в блок try...catch и отследить исключение. Однако область работы фильтров исключения несколько шире. Они позволяют отследить не только исключения, возникающие в самом методе, но исключения, генерируемые результатами действий, а также другими применяемыми к данному действию фильтрами. В этом и состоит мощь данного типа фильтров.

Все фильтры исключений должны применять интерфейс **IExceptionFilter**:

|  |  |
| --- | --- |
| 1  2  3  4 | public interface IExceptionFilter  {      void OnException(ExceptionContext filterContext);  } |

И если вдруг приложение выбрасывает необрабатываемое исключение, то фильтр вызывает метод OnException.

Передаваемый в этот метод параметр - ExceptionContext является объектом, производным от ControllerContext. Поэтому из него можно извлечь как специфичную для фильтра информацию, так и общую информацию о запросе.

В частности класс ExceptionContext обладает следующими свойствами, которые позволяют получить информацию об исключении:

|  |  |  |
| --- | --- | --- |
| **Свойство** | **Тип** | **Описание** |
| ActionDescriptor | ActionDescriptor | Содержит информацию о методе действия, на котором было выброшено исключение |
| Exception | Exception | Представляет само необработанное исключение |
| ExceptionHandled | bool | Значение, показывающее, считается ли исключение обработанным. Если мы на фильтре помечаем его значение в true, то исключение считается обработанным |
| Result | ActionResult | Результат метода действия, к которому применяется фильтр исключения |

С помощью свойства Exception мы можем получить доступ к выбрасываемому исключению.

Установив свойство ExceptionHandled в true, фильтр тем самым помечает исключение как обработанное.

С помощью свойства Result фильтр управляет результатом действий. Общераспространенной практикой в данном случае является перенаправление пользователя на страницу ошибки или отображение ошибки на экране.

Теперь создадим простенький фильтр, который будет обрабатывать исключение IndexOutOfRangeException:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.Web.Mvc;    namespace Filters.Filters  {      public class IndexException : FilterAttribute, IExceptionFilter      {            public void OnException(ExceptionContext exceptionContext)          {              if (!exceptionContext.ExceptionHandled && exceptionContext.Exception is IndexOutOfRangeException)              {                  exceptionContext.Result = new RedirectResult("/Content/ExceptionFound.html");                  exceptionContext.ExceptionHandled = true;              }          }      }  } |

Здесь в методе OnException первым делом мы проверяем, не установлено ли значение свойства ExceptionHandled. Если оно установлено в true, следовательно, какой-то другой фильтр исключений уже обработал данное исключение. Также проверяется тип исключения. Поскольку мы ловим в данном случае только исключения типа IndexOutOfRangeException, следовательно, нас только они интересуют.

Далее мы устанавливаем результат метода, к которому применен фильтр. Предполагается, что в проекте в каталоге Content у нас находится некоторая страница *ExceptionFound.html*, которая отображает пользователю сообщение об ошибке.

В данном случае вадно пометить исключение как обработанное: exceptionContext.ExceptionHandled = true. Иначе, если мы этого не сделаем, то мы можем увидеть в браузере все диагностическое сообщение об ошибке, которое обычно посылает фреймворк в ответ клиенту.

Пример использования:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | [IndexException]  public ActionResult Index()  {      int[] mas = new int[2];      mas[6] = 4;      return View();  } |

В данном случае метод Index выбросит необработанное исключение, и оно будет объектом типа NullReferenceException, а пользователь будет перенаправлен на страницу *ExceptionFound.htm*.

Подобным образом мы можем обработать и другие типы исключений.

### HandleErrorAttribute. Встроенная обработка исключений.

Создавать свои фильтры исключений необязательно, так как во фреймворке имеется встроенная реализация интерфейса IExceptionFilter - атрибут HandleErrorAttribute. Он имеет ряд свойств, с помощью которых мы можем произвести гибкую настройку фильтра:

|  |  |  |
| --- | --- | --- |
| **Свойство** | **Тип** | **Описание** |
| ExceptionType | Type | Представляет тип обрабатываемого исключения. По умолчанию используется System.Exception |
| View | string | Имя представления, которое рендерится данным фильтром. Если значение не задано, то по умолчанию используются следующие пути: */Views/Имя\_контроллера/Error.cshtml* или */Views/Shared/Error.cshtml* |
| Master | string | Имя используемой мастер-страницы |

При обработке исключения фильтр исключений посылает статусный код HTTP 500 и генерирует указанное в свойстве View представление. Например, используем предыдущий пример с фильтром исключений, применив встроенную реализацию:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | [HandleError(ExceptionType = typeof(System.IndexOutOfRangeException), View = "ExceptionFound")]  public ActionResult About()  {      int[] mas = new int[2];      mas[6] = 4;      return View();  } |

В данном случае очевидно, что на строке mas[6] = 4; будет выброшено исключение. В режиме отладки у вас приостановится выполнение программы, тогда вы можете нажать на кнопку Continue на панели инструментов. Здесь опять мы обрабатываем исключение типа IndexOutOfRangeException, и при возникновении такового генерируем в ответ представление *ExceptionFound.cshtml*. Данное представление должно находиться в проекте в каталоге *Views/Имя\_контроллера/*.

Сразу надо сказать, что если вы хотите при разработке видеть обрабатываемые фильтром HandleErrorAttribute, то надо включить данную функциональность в файле конфигурации *web.config* с помощью тега <customErrors mode="On" />:

|  |  |
| --- | --- |
| 1  2  3  4  5 | <system.web>      <customErrors mode="On"  />        <!-- далее остальное содержание узла system.web -->  </system.web> |

## Фильтры действий и результатов

### Фильтр действий

Фильтры действий позволяют нам проконтролировать входной контекст запроса при доступе к действию, а также выполнить определенные действия по завершению работы метода действий. Например, мы можем изменить выходной результат метода.

Фильтр действий должен реализовать интерфейс IActionFilter:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public interface IActionFilter  {      void OnActionExecuting(ActionExecutingContext filterContext);      void OnActionExecuted(ActionExecutedContext filterContext);  } |

Либо мы должны унаследовать новый фильтр от абстрактного класса ActionFilterAttribute, который уже реализует интерфейс IResultFilter:

|  |  |
| --- | --- |
| 1 | public abstract class ActionFilterAttribute : FilterAttribute, IActionFilter, IResultFilter |

Метод OnActionExecuting вызывается перед вызовом метода действий. А метод OnActionExecuted - после.

С помощью метода OnActionExecuting можно получить информацию о запросе, изменить ее либо вовсе отменить запрос. Передаваемый в метод объект ActionExecutingContext является производным от класса ControllerContext и определяет следующие два дополнительных свойства:

|  |  |  |
| --- | --- | --- |
| **Свойство** | **Тип** | **Описание** |
| ActionDescriptor | ActionDescriptor | Предоставляет информацию о вызываемом методе действия |
| Result | ActionResult | Результат метода действий |

С помощью свойства Result мы можем отменить результат действия, присвоив этому свойству другой результат действия.

В метод OnActionExecuted в качестве параметра передается объект ActionExecutedContext, который также является производным от классаControllerContext. Он определяет некоторые дополнительные свойства:

|  |  |  |
| --- | --- | --- |
| **Свойство** | **Тип** | **Описание** |
| ActionDescriptor | ActionDescriptor | Предоставляет информацию о вызываемом методе действий |
| Canceled | bool | Хранит значение, показывающее, отменен ли вызов действия. Если имеет значение true, если вызов действия был отменен другим фильтром |
| Exception | Exception | Возвращает исключение, выбрасываемое данным методом действий или другим фильтром |
| ExceptionHandled | bool | Хранит значение, показывающее, обработано ли исключение. Если имеет true, если исключение обработано |
| Result | ActionResult | Результат метода действий |

В качестве примера мы можем создать следующий фильтр действия:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public class MyActionAttribute : FilterAttribute, IActionFilter  {      public void OnActionExecuted(ActionExecutedContext filterContext)      {          filterContext.HttpContext.Response.Write("Действие выполнено");      }        public void OnActionExecuting(ActionExecutingContext filterContext)      {          if (filterContext.HttpContext.Request.Browser.Browser=="Opera")          {              filterContext.Result = new HttpNotFoundResult();          }      }  } |

И использование:

|  |  |
| --- | --- |
| 1  2  3  4  5 | [MyAction]  public ActionResult Index()  {      return View();  } |

Здесь в методе OnActionExecuting в случае, если клиент использует браузер Opera (например, мы хотим запретить поддержку своего сайта для этого браузера), то мы возвращаем клиенту статусный код 404. А в методе OnActionExecuted мы добавляем в выходной поток строку.

Примером стандартной реализации фильтров действий может служить фильтр **InitializeSimpleMembershipAttribute**, добавляемый по умолчанию в проект по шаблону Internet Application. Этот фильтр используется для начальной настройки членства пользователей и ролей и применяется к методам контроллера AccountController (в шаблоне Internet Application):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | [AttributeUsage(AttributeTargets.Class | AttributeTargets.Method, AllowMultiple = false, Inherited = true)]  public sealed class InitializeSimpleMembershipAttribute : ActionFilterAttribute  {      private static SimpleMembershipInitializer \_initializer;      private static object \_initializerLock = new object();      private static bool \_isInitialized;        public override void OnActionExecuting(ActionExecutingContext filterContext)      {          // Ensure ASP.NET Simple Membership is initialized only once per app start          LazyInitializer.EnsureInitialized(ref \_initializer, ref \_isInitialized, ref \_initializerLock);      }        private class SimpleMembershipInitializer      {          public SimpleMembershipInitializer()          {              Database.SetInitializer<UsersContext>(null);                try              {                  using (var context = new UsersContext())                  {                      if (!context.Database.Exists())                      {                          // Create the SimpleMembership database without Entity Framework migration schema                          ((IObjectContextAdapter)context).ObjectContext.CreateDatabase();                      }                  }                    WebSecurity.InitializeDatabaseConnection("DefaultConnection", "Users", "Id", "Email", autoCreateTables: true);              }              catch (Exception ex)              {                  throw new InvalidOperationException("The ASP.NET Simple Membership database could not be initialized.                          For more information, please see http://go.microsoft.com/fwlink/?LinkId=256588", ex);              }          }      }  } |

### Фильтр результатов

Фильтры результатов во многом похожи на фильтры действий, поскольку так же могут срабатывать как до возвращения результата действия, так и после. Фильтры результатов реализуют интерфейс IResultFilter:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public interface IResultFilter  {      void OnResultExecuting(ResultExecutingContext filterContext);      void OnResultExecuted(ResultExecutedContext filterContext);  } |

Метод OnResultExecuting вызывается перед тем, как метод действия начнет возвращать результат действия. А метод OnResultExecuted - после того, как метод действия возвратил результат. Эти методы в качестве параметров принимают объекты ResultExecutingContext иResultExecutedContext, которые аналогичны объектам OnActionExecuting и OnActionExecuted соответственно.

Например, мы можем выводить в выходной поток имя пользователя (если он авторизован) и текущее время запроса к серверу:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | using System;  using System.Web;  using System.Web.Mvc;  namespace Filters.Filters  {      public class MyResultAttribute: FilterAttribute, IResultFilter      {            public void OnResultExecuted(ResultExecutedContext filterContext)          {                filterContext.HttpContext.Response.Write("Время текущего запроса HTTP: " + filterContext.HttpContext.Timestamp);          }            public void OnResultExecuting(ResultExecutingContext filterContext)          {                filterContext.HttpContext.Response.Write("Текущий пользователь: " + filterContext.HttpContext.User.Identity.Name);          }      }  } |

### Встроенная реализация фильтров действий и фильтров результатов

Как говорилось в начале главы, фильтры действий и фильтры результатов объединены в одну реализацию - абстрактный класс**ActionFilterAttribute**, который объединяет черты обоих фильтров:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public abstract class ActionFilterAttribute : FilterAttribute, IActionFilter, IResultFilter  {      public virtual void OnActionExecuting(ActionExecutingContext filterContext){}      public virtual void OnActionExecuted(ActionExecutedContext filterContext){}      public virtual void OnResultExecuting(ResultExecutingContext filterContext) {}      public virtual void OnResultExecuted(ResultExecutedContext filterContext) {}  } |

Мы можем реализовать только те методы, которые нам нужны, либо сразу все, что делает данный класс довольно гибким.

## Глобальные фильтры

Мы посмотрели на различные виды фильтров. Как вы видели, мы можем применить фильтры как к отдельным действиям, так и ко всему контроллеру - то есть ко всем его действиям. Однако если мы хотим применить фильтр ко всем контроллерам, а контроллеров у нас много? В данном случае в mvc реализована функциональность глобальных фильтров. Эти фильтры применяются ко всем действиям всех контроллеров.

Откроем файл *FilterConfig.cs*, который находится в папке *App\_Start*. В этом файле определен класс **FilterConfig**, который в методе**RegistreGlobalFilters** осуществляет регистрацию глобальных фильтров в коллекцию filters. Эта коллекция представляет собой объектGlobalFilterCollection, который передается в качестве параметра при вызове метода в файле Global.asax.

Итак, мы можем зарегистрировать глобальные фильтры так:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public class FilterConfig  {      public static void RegisterGlobalFilters(GlobalFilterCollection filters)      {          filters.Add(new HandleErrorAttribute());            filters.Add(new MyExceptionAttribute());          filters.Add(new MyResultAttribute());      }  } |

## Дополнительные встроенные фильтры

### Фильтр RequireHttps

Фильтр RequireHttps заставляет использовать протокол HTTPS, а браузер перенаправит пользователя на то же действие, только с префиксом https.

Данный фильтр применяется только к GET-запросам.

### Фильтр OutputCache

Данный фильтр говорит mvc-фреймворку кэшировать вывод метода действия, чтобы полученный контент можно было в дальнейшем использовать повторно.

Кэширование может увеличить производительность, особенно когда идет речь о выборке из базы данных, которая может занимать значительное время.

С помощью параметра Duration мы можем настроить время (в секундах):

|  |  |
| --- | --- |
| 1  2  3  4  5 | [OutputCache (Duration=360)]  public ActionResult Index()  {      //.............  } |

### Атрибут ValidateAntiforgeryToken

Фильтр ValidateAntiforgeryToken предназначен для противодействия подделке межсайтовых запросов, производя верификацию токенов при обращении к методу действия. Наиболее частым случаем является применение данного фильтра к методам, отвечающим за авторизацию, например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | [ValidateAntiForgeryToken]  public ActionResult Login(LoginModel model, string returnUrl)  {      if (ModelState.IsValid && WebSecurity.Login(model.Email, model.Password, persistCookie: model.RememberMe))      {          return RedirectToLocal(returnUrl);      }        ModelState.AddModelError("", "Неверный пароль или логин");      return View(model);  } |

# Привязка модели

## Введение в привязку моделей

В предыдущих главах мы рассмотрели модели и узнали, как передавать в методы контроллера значения из представлений. Но за кадром остался весь механизм, который способствует сопоставлению значений с используемыми в методах контроллера параметрами. И этот механизм называется *привязкой модели*.

Чтобы понять смысл привязки, посмотрим на примере. Допустим, мы отправляем на сервер форму с некоторыми данными для модели Book. И в некотором методе контроллера мы можем получить переданные данные из формы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | [HttpPost]  public ActionResult Edit()  {      Book b = new Book();      b.Name = Request.Form["Name"];      b.Author = Request.Form["Author"];      // .........  } |

Однако свойств модели Book, которые надо получить из запроса, может быть множество. Кроме того, вы можете заметить, что поля для ввода свойств модели имеют имя (атрибут name), совпадающее с именем свойства модели. Чтобы облегчить процесс получения присланных данных и был создан механизм привязки. Благодаря чему мы можем просто написать:

|  |  |
| --- | --- |
| 1  2  3  4  5 | [HttpPost]  public ActionResult Edit(Book b)  {      //.........  } |

При этом привязка модели не ограничена POST-запросами и сложными параметрами типа объекта Book. Привязка может также работать и с более простыми параметрами, например со следующим действием Edit, обрабатывающим HTTP-запрос GET:

|  |  |
| --- | --- |
| 1  2  3  4 | public ActionResult Edit(int id)  {      // ...  } |

### Привязчики моделей

После сопоставления запроса с некоторым маршрутом контролллеров вызываются специальные компоненты - **активаторы действий**(action invoker), которые вызывают нужное действие контроллера и передают в него значения из контекста запроса. А чтобы сопоставить полученные значения с конкретными параметрами активаторы действий, в частности, активатор по умолчанию - ControllerActionInvoker, используют **привязчики модели (model binder)**. Привязчики моделей и осуществляют собственно привязку.

Все привязчики моделей должны реализовать интерфейс **IModelBinder**:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public interface IModelBinder  {      object BindModel (ControllerContext controllerContext,                          ModelBindingContext bindingContext);  } |

В принципе для каждого отдельного типа может существовать свой привязчик модели. При просмотре параметров метода действия активатор действий ищет для каждого типа параметра соответствующий привязчик и вызывает его метод BindModel. В случае, если соответствующего данному типу привязчика не обнаружится, то используется привязчик по умолчанию - DefaultModelBinder.

Затем привязчик использует специальные компоненты - **поставщики значений (value provider)** для поиска значений в различных частях запроса.

## DefaultModelBinder

Привязчик **DefaultModelBinder** используется по умолчанию, если для данного типа не определен другой привязчик. Чтобы получить значения для параметров, привязчик просматривает следующие объекты строго по порядку:

1. Request.Form. В данном случае привязчик получает значения, отправленные с помощью форм
2. RouteData.Values. Здесь получает значения с помощью маршрутов приложения
3. Request.QueryString. В данном случае привязчик извлекает значения из строки запроса
4. Request.Files. В данном случае используются загруженные на сервер файлы

Если в одном из этих объектов будет обнаружен соответствующий элемент, то на нем поиск прекращается, а параметру передается найденное значение. Например, привязчик ищет значения для параметров следующего действия:

|  |  |
| --- | --- |
| 1  2  3  4 | public ActionResult Edit(int id)  {      // ...  } |

Чтобы DefaultModelBinder мог связать значение с параметром, элемент данных запроса должен обязательно иметь то же имя, что и параметр. То есть в данном случае они оба должны иметь имя id.

Итак, привязчик DefaultModelBinder просматривает в поиске значения для параметра id следующие пути:

1. Request.Form["id"]
2. RouteData.Values["id"]
3. Request.QueryString["id"]
4. Request.Files["id"]

При привязке простых типов DefaultModelBinder преобразует строковое выражение параметра из данных запроса к типу параметра с помощью класса System.ComponentModel.TypeDescriptor. Если же значение нельзя преобразовать, то привязка оканчивается неудачей. Поэтому есть смысл использовать параметры, допускающие значение null, а в метод действия вкладывать соответствующую логику по проверке значения:

|  |  |
| --- | --- |
| 1  2  3  4 | public ActionResult Edit(int? id)  {      // ...  } |

При получении значений для сложных типов, например, для той же модели Book, DefaultModelBinder с помощью рефлексии просматривает объект и находит все свойства объекта, которые могут использоваться при привязке. Например, DefaultModelBinder видит, что объект Bookимеет свойство Author, то привязчик будет искать в запросе параметр "Author".

Если свойство представляет простой тип, то для него используется, как было указано выше преобразование с помощью класса классаSystem.ComponentModel.TypeDescriptor. Если же свойство представляет сложный тип - то опять повторяется рефлексия и весь цикл.

### Выборочная привязка

Иногда возникает возможность исключить некоторые свойства их привязки модели. Мы можем это сделать с помощью атрибута Bind.

Для включения только определенных свойств мы можем использовать свойство Include данного атрибута:

|  |  |
| --- | --- |
| 1  2  3  4 | public ActionResult Create([Bind (Include="Name, Author")] Book b)  {      // ...  } |

В данном случае мы указываем, что в привязке будут участвовать только свойства Name и Author. Остальные же свойства из привязки исключаются.

Либо мы можем использовать свойство Exclude атрибута Bind, чтобы исключить свойство из привязки:

|  |  |
| --- | --- |
| 1  2  3  4 | public ActionResult Create([Bind (Exclude="Year")] Book b)  {      // ...  } |

В данном случае выборочная привязки применена к методу Create. Что если нам надо осуществить выборочную привязку глобально во всем приложении? Тогда мы можем применить атрибут непосредственно к модели, и в этом случае атрибут будет применен по умолчанию ко всем методам действий контроллеров проекта

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | [Bind (Exclude="Year")]  public class Book  {      [ScaffoldColumn(false)]      public int Id { get; set; }        [Required (ErrorMessage="Поле должно быть установлено")]      [Display(Name = "Название")]      public string Name { get; set; }        [Required]      [Display(Name = "Автор")]      public string Author { get; set; }        [Display(Name = "Год")]      public int Year { get; set; }  } |

## Явная привязка модели

При использовании параметра в методе действия привязка модели работает неявно. Но мы можем вызвать на контроллере и явную привязку модели с помощью методов **UpdateModel** и **TryUpdateModel**. Если модель не прошла валидацию, то метод UpdateModelвыбрасывает исключение. Ниже показан пример использования метода UpdateModel в действии Edit вместо применения параметра:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | [HttpPost]  public ActionResult Edit()  {      var book = new Book();      try      {          UpdateModel(book);          db.Entry(book).State = EntityState.Modified;          db.SaveChanges();          return RedirectToAction("Index");      }      catch      {          ViewBag.Message="Во время редактирования возникли ошибки";          return RedirectToAction("Index");      }  } |

TryUpdateModel также вызывает привязку модели, но не выбрасывает исключение. Этот метод возвращает значение типа bool - если это значение равно true, модель прошла привязку, если false, то валидация прошла неудачно.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | [HttpPost]  public ActionResult Edit()  {      var book = new Book();      if (TryUpdateModel(book))      {          db.Entry(book).State = EntityState.Modified;          db.SaveChanges();          return RedirectToAction("Index");      }      else      {          ViewBag.Message="Во время редактирования возникли ошибки";          return RedirectToAction("Index");      }  } |

## Поставщики значений

Как уже говорилось в начале главы, для поиска значений привязчик модели использует специальные компоненты - **поставщики значений (value providers)**. Поставщики значений предоставляют доступ к информации, которая затем используется в привязке модели.

Поставщики значений поставляются **фабриками поставщиков значений** (value provider factories) и системой поиска данных для этих поставщиков значений в том порядке, в котором они зарегистрированы (регистрация по принципу очереди - вверху первый, внизу последний). Разработчики могут создавать свои собственные фабрики и поставщики значений и добавлять их в список фабрикValueProviderFactories.Factories. Для предоставления дополнительного источника данных для привязки модели разработчики могут применить фабрику или поставщик значений.

Все поставщики значений реализуют интерфейс **IValueProvider**:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public interface IValueProvider  {      bool ContainPrefix (string prefix);      ValueProviderResult GetValue (string key);  } |

В конкретных реализациях метод ContainPrefix сопоставляет некоторые данные с указанным префиксом, а метод GetValue получает данные для определенного ключа.

Создадим свой поставщик значений. Пусть, к примеру, он будет получать для нас название браузера пользователя. Определим поставщик:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class BrowserValueProvider : IValueProvider  {      public bool ContainsPrefix(string prefix)      {          return string.Compare("browser", prefix, true) == 0;      }        public ValueProviderResult GetValue(string key)      {          return ContainsPrefix(key) ? new ValueProviderResult("Ваш браузер: "+HttpContext.Current.Request.Browser.Browser, null,              CultureInfo.InvariantCulture) : null;      }  } |

В методе GetValue, если ключ совпадает с префиксом, то мы возвращаем объект ValueProviderResult, в конструктор которого передаем три параметра - первый параметр принимает значение, ассоциируемое с ключом. Второй параметр используется для отслеживания ошибок, А в третьем параметре передается информация о культуре, с которое ассоциируется значение. В данном случае второй и третий параметр нам не столь важны.

Как уже писалось выше, чтобы задействовать свой поставщик значений, нам надо определить фабрику поставщика:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class BrowserValueProviderFactory : ValueProviderFactory  {      public override IValueProvider GetValueProvider(ControllerContext controllerContext)      {          return new BrowserValueProvider();      }  } |

И в конце нам надо зарегистрировать фабрику поставщика значений в файле Global.asax:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public class MvcApplication : System.Web.HttpApplication  {      protected void Application\_Start()      {          ValueProviderFactories.Factories.Add(new BrowserValueProviderFactory());            //......................................................      }  } |

В данном случае мы добавляем наш поставщик в конец коллекции ValueProviderFactories.Factories. Фреймворк просматривает последовательно все фабрики поставщиков значений в порядке регистрации. Если будет найден соответствующий поставщик, то следующие после него уже не используются.

В нашем случае это не имеет значения. Однако если возникнет необходимость поставить свою фабрику поставщика значений на первое место в списке, то мы можем сделать это следующим образом: ValueProviderFactories.Factories.Insert(0, new BrowserValueProviderFactory());

Теперь мы можем определить метод действия, в котором протестируем работу поставщика значений:

|  |  |
| --- | --- |
| 1  2  3  4 | public string BrowserInfo(string browser)  {      return browser;  } |

Итак, отправив запрос данному методу действий, мы получим в качестве ответа название нашего браузера.

Как вы видите, мы не оправляем в форме или в строке запроса параметр browser и его значение. Его находит сам наш поставщик.

## Создание привязчика модели

Чтобы создать привязчик модели, нам надо реализовать интерфейс **IModelBinder**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | public class BookModelBinder : IModelBinder  {      public object BindModel(ControllerContext controllerContext, ModelBindingContext bindingContext)      {          // Получаем поставщик значений          var valueProvider = bindingContext.ValueProvider;            // получаем данные по одному полю          ValueProviderResult vprId = valueProvider.GetValue("Id");            // получаем данные по остальным полям          string name = (string)valueProvider.GetValue("Name").ConvertTo(typeof(string));          string author = (string)valueProvider.GetValue("Author").ConvertTo(typeof(string));          int year = (int)valueProvider.GetValue("Year").ConvertTo(typeof(int));          Book book = new Book() { Name = name + " (new)", Author = author, Year = year };            // если поле Id определено (редактирование)          if (vprId != null)          {              book.Name = name; // без new              book.Id=(int)vprId.ConvertTo(typeof(int));          }          return book;      }  } |

Данный интерфейс содержит единственный метод BindModel, который мы и реализуем в новом классе.

Сначала мы получаем поставщик значений, а затем с его помощью получаем значения для отдельных свойств будущего объекта Book. Чтобы корректным образом получить значение, мы преобразуем его к надлежащему типу: int id = (int)valueProvider.GetValue("Id").ConvertTo(typeof(int));. И в конце возвращаем созданный объект Book.

Теперь нам надо зарегистрировать привязчик модели. Это можно сделать двумя способами. Первый способ - регистрация в файле Global.asax:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public class MvcApplication : System.Web.HttpApplication  {      protected void Application\_Start()      {          ModelBinders.Binders.Add(typeof(Book), new BookModelBinder());            //......................................................      }  } |

Альтернативный способ регистрации привязчика - использование соответствующего атрибута для модели:

|  |  |
| --- | --- |
| 1  2  3  4  5 | [ModelBinder(typeof(BookModelBinder))]  public class Book  {      //...........  } |

Теперь для модели Book будет использоваться привязчик BookModelBinder. Ну и в любом методе, который принимает в качестве параметра объект Book, можно протестировать привязчик:

|  |  |
| --- | --- |
| 1  2  3  4  5 | [HttpPost]  public ActionResult Index(Book book)  {      //...........  } |

# jQuery и AJAX

## Использование JavaScript/jQuery

Современные веб-приложения практически невозможно представить без языка клиентской части - JavaScript. Даже при использовании таких серверных языков и технологий, как PHP, ASP.NET, трудно обойтись без JavaScript. Однако чистый JavaScript в реальности используется все меньше. Ему на смену приходят специальные библиотеки, в частности jQuery. Применительно к ASP.NET MVC при создании веб-приложений библиотеки jQuery играют очень большую роль.

### Подключение файлов JavaScript/jQuery

По умолчанию все проекты, кроме проектов по шаблону Empty, уже содержат необходимый набор скриптов, в том числе библиотеки jQuery:

Как вы видите, большинство скриптов имеют свои двойники с суффиксом *min*, например, *jquery-1.7.1.js* и *jquery-1.7.1.min.js*. Оба скрипта представляют одну и ту же функциональность. Но вторая версия представляет минимизированную версию (поэтому и идет с суффиксом*min*). Зачем она нужна? Минимизированные скрипты гораздо меньше по объему (иногда даже на 60-70%), поэтому их предпочтительнее использовать в реальных приложениях, так как пользователь тратит меньше времени и трафика на их загрузку. В то же время их не очень удобно читать. Поэтому для большего удобства разработчиков полные и минимизированные скрипты идут вместе.

Вкратце посмотрим, зачем нужны большинство скриптов, идущих по умолчанию:

* *jquery-1.7.1.js* - базовая библиотека jQuery, на которую опираются большинство других скриптов. В данном случае используется версия 1.7.1. Однако библиотека постоянно обновляется, поэтому можно использовать более новые версии.
* *jquery-ui-1.8.20.js* - библиотека jQuery UI, которая включает различные виджеты, предназначенные для создания пользовательского интерфейса
* *jquery.unobtrusive-ajax.js* - представляет функциональность для ненавязчивого JavaScript
* *jquery.validate.js* - представляет функционал для валидации на стороне клиента
* *jquery.validate.unobtrusive.js* - предоставляет поддержку ненавязчивой валидации
* *jquery-1.7.1-vsdoc.js* и *jquery.validate-vsdoc.js* - используются для поддержки документации и IntelliSense по соответствующим библиотекам в Visual Studio

Чтобы подключить файл javascript используется метод Render класса System.Web.Optimization.Scripts:@Scripts.Render("~/scripts/jquery.unobtrusive-ajax.js")

Этот метод принимает в качестве параметра строку - полный путь к скрипту.

Либо мы можем использовать, например, хелпер Url.Content : <script src="@Url.Content("~/scripts/jquery.validate.unobtrusive.js")" type="text/javascript"></script>

Если файлы javascript будут использоваться большинством представлений приложения, то удобнее всего добавить их сразу на мастер-страницу.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <meta name="viewport" content="width=device-width" />      <title>@ViewBag.Title</title>      @Styles.Render("~/Content/css")      @Scripts.Render("~/scripts/jquery-1.7.1.js")      @Scripts.Render("~/scripts/jquery.unobtrusive-ajax.js")  </head>  <body>      @RenderBody()      @RenderSection("scripts", required: false)  </body>  </html> |

## Краткий обзор jQuery

Библиотека jQuery предназначена прежде всего для удобного поиска и манипулирования элементами на веб-странице. При нахождении определенного элемента с помощью jQuery можно повесить на него обработчики событий, анимировать, скрыть или, наоборот, отобразить, создать для элемента интерактивное взаимодействие с пользователем, изменить его стили и т.д. И даже если вы не работали раньше с jQuery, ее освоение не составит особых трудностей. Для более подробного ознакомления с данной библиотекой можно обратиться к руководству [Изучаем jQuery](http://metanit.com/web/jquery/index.php). Здесь же я представлю краткий обзор возможностей библиотеки.

### Функция jQuery

Функция jQuery позволяет использовать всю мощь библиотеки jQuery. Данная функция (jQuery) в качестве псевдонима имеет знак $ (так как символ $ легко набрать на клавиатуре, и он представляет действительное имя функции, которое можно употреблять в JavaScript). Поэтому следующие записи функции jQuery будут идентичны:

|  |  |
| --- | --- |
| 1  2  3 | jQuery(document).ready(function(){    // здесь код анонимной функции  }); |
| 1  2  3 | $(function(){    // здесь код анонимной функции  }); |

|  |  |
| --- | --- |
| 1  2  3 | $(document).ready(function(){    // здесь код анонимной функции  }); |

Все три вышеописанных случая идентичны и срабатывают сразу после загрузки веб-страницы. Весь остальной функционал помещается внутрь функции jQuery. И поскольку данная функция осуществляет выборку и модификацию элементов, то для нее иногда необходимо, чтобы вся веб-страница была уже загружена. Поэтому скрипт данной функции или ссылку на файл скрипта помещают обычно в самый низ веб-страницы.

### Селекторы jQuery

Для выборки из структуры страницы нужных элементов используются *селекторы*.

##### Основные селекторы jQuery

|  |  |  |
| --- | --- | --- |
| **Шаблон селектора** | **Значение** | **Пример** |
| $("Element") | Выбирает все элементы с данным именем тега | $("p") выбирает все теги p.  $("ul") выбирает все элементы ul |
| $("#id") | Выбирает элемент с определенным значением id | Так, в следующем коде:  **<div id="box1"></div>**  <div id="box2"></div>  селектор $("#box1") выбирает элемент, помеченный жирным |
| $(".className") | Выбирает все элементы с определенным значением атрибута class | Допустим, у нас следующий код:  **<div class="apple"></div>**  **<div class="apple"></div>**  <div class="orange"></div>  <div class="banana"></div>  то селектор $(".apple") выбирает все элементы, помеченные жирным |
| $("selector1,selector2,selectorN") | Выбирает элементы, которые соответствуют данным селекторам | Если у нас следующий код:  **<div class="apple"></div>**  **<div class="apple"></div>**  **<div class="orange"></div>**  <div class="banana"></div>  то селектор $(".apple, .orange") выберет элементы, выделенные жирным |

Например:

|  |  |
| --- | --- |
| 1  2  3  4 | $(function () {      $(".results").css("top", "20px");      });  }); |

Сначала мы получаем элемент, у которого class имеет значение **results** (< div class="results"></div>), а потом с помощью функции cssустанавливаем определенное значение для его свойства top. Причем если у нас на странице несколько элементов, у которыхclass="results", то селектор вернет весь набор из этих элементов. И к каждому из элементов данного набора будет применено преобразование.

Выше в таблице показан лишь небольшой базовый список селекторов. Полный же список селекторов вы можете найти на сайте<http://www.w3.org/TR/css3-selectors/>

### Фильтры jQuery

В дополнение к селекторам применяются фильтры. Можно выделить следующий набор базовых фильтров:

|  |  |
| --- | --- |
| **Фильтр** | **Значение** |
| :eq(n) | Выбирает n-й элемент выборки (нумерация начинается с нуля) |
| :even | Выбирает элементы с четными номерами |
| :odd | Выбирает элементы с нечетными номерами |
| :first | Выбирает первый элемент выборки |
| :last | Выбирает последний элемент выборки |
| :gt(n) | Выбирает все элементы с номером, большим n |
| :lt(n) | Выбирает все элементы с номером, меньшим n |
| :header | Выбирает все заголовки (h1, h2, h3) |
| :not(селектор) | Выбирает все элементы, которые не соответствуют селектору, указанному в скобках |

Например, если у нас на странице несколько элементов, у которых class="results", а нам надо выбрать только первый, то мы можем применить следующие выражения: $(".results:first") или $(".results:eq(0)")

Специальный род фильтров - фильтры контента обеспечивают доступ к элементам, имеющим определенное содержимое:

|  |  |
| --- | --- |
| **Фильтр** | **Значение** |
| :contains('content') | Получает все элементы, которые содержат content |
| :has('селектор') | Получает все элементы, которые содержат хотя бы один дочерний элемент, соответствующий селектору |
| :empty | Получает все элементы, которые не имеют дочерних элементов |
| :first-child | Получает все элементы, которые являются первыми дочерними элементами в своих родителях |
| :last-child | Получает все элементы, которые являются последними дочерними элементами в своих родителях |
| :nth-child(n) | Получает все элементы, которые являются n-ными элементами в своих родителях (нумерация идет с единицы) |
| :only-child | Получает все элементы, которые являются единственными дочерними элементами в своих родителях |
| :parent | Получает все элементы, которые имеют, как минимум, один дочерний элемент |

Например, если мы хотим получить все элементы, содержащие текст asp.net mvc, мы можем применить следующее выражение:$(':contains("asp.net mvc")')

И завершая обзор фильтров, следует упомянуть о фильтрах форм, которые позволяют получить определенные элементы html-форм:

|  |  |
| --- | --- |
| **Фильтр** | **Значение** |
| button | Получает все элементы button и элементы input с типом button |
| :checkbox | Получает все элементы checkbox |
| :checked | Получает все отмеченные элементы checkbox и radio |
| :disabled | Получает все элементы, которые отключены |
| :enabled | Получает все элементы, которые включены |
| :input | Получает все элементы input |
| :password | Получает все элементы password |
| :radio | Получает все элементы radio |
| :reset | Получает все элементы reset |
| :selected | Получает все отмеченные элементы option |
| :submit | Получает все элементы input с типом submit |
| :text | Получает все элементы input с типом text |

Мы можем комбинировать в одном выражении несколько селекторов и фильтров: $('.results:odd:has('img')'). В данном случае мы выбираем все нечетные элементы с class="results", которые содержат элементы img, то есть изображения.

### События jQuery

jQuery предоставляет специальные методы для распространенных событий, как например, click или submit. Мы можем повесить свои обработчики для событий mouseover (наведение мыши) или keydown(нажатие клавиатуры) на любой элемент веб-страницы.

Например, обработчик нажатия мыши на элемент с id="bg" мог бы выглядеть следующим образом.

|  |  |
| --- | --- |
| 1 | $("#bg").mousedown (function (e) {}}); |

Или для примера обработаем нажатие клавиши:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | $(document).keydown(function(e){      // если нажата клавиша вверх       if (e.which==38)       {          // поднимаем некоторый элемент на 5 пикселей вверх          var top = parseInt($("#paddleB").css("top"));          $("#paddleB").css("top", top-5);      }  }); |

### Методы jQuery

Как говорилось выше, jQuery выполняет две основные задачи - поиск элементов и их изменение. Если для поиска предназначены селекторы и фильтры, то для манипуляции над элементами используются методы. Эти методы позволяют изменять внешний вид элемента, анимировать его, перемещать в структуре элементов DOM. Это методы довольно многочисленны, поэтому рассмотрим лишь вкратце:

|  |  |
| --- | --- |
| **Метод** | **Описание** |
| addClass('someClass') | Добавляет для выбранного элемента класс someClass |
| removeClass('someClass') | Удаляет для выбранного элемента класс someClass |
| toggleClass('someClass') | Переключает для выбранного элемента класс someClass - если его нет, он добавляется, а если он есть - то удаляется |
| css('свойство', 'значение') | Устанавливает для указанного свойства выбранного элемента указанное значение ($("#paddleB").css("top",25);) |
| append('новый элемент') | Вставляет внутрь выбранного элемента новый элемент в качестве последнего дочернего ($("#results").append('<li>Новый элемент списка</li>');) |
| prepend('новый элемент') | Вставляет внутрь выбраного элемента новый элемент в качестве первого дочернего |
| empty() | Удаляет все дочерние элементы у выбранного элемента |
| remove() | Удаляет элемент из структуры элементов DOM |
| attr('атрибут','значение') | Устанавливает для атрибута новое значение |
| removeAttr('атрибут') | Удаляет атрибут у выбранных элементов |
| children() | Получает все дочерние элементы у выбранных элементов |
| parent() | Получает все родительские элементы у выбранных элементов |
| parent() | Получает все родительские элементы у выбранных элементов |
| hide() | Скрывает выбранные элементы |
| show() | Отображает выбранные элементы |
| toggle() | Скрывает видимые элементы и отображает невидимые |
| animate() | Анимирует элемент |

Например, стандартный прием, когда по наведению курсора мыши на изображение, оно увеличивается в размерах, а после отвода курсора - уменьшается:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | $(function () {      $("img").mouseover(function () {          $(this).animate({ height: '+=20', width: '+=20' });      });      $("img").mouseout(function () {          $(this).animate({ height: '-=20', width: '-=20' });      });  }); |

Сначала при помощи селектора мы выбираем все элементы img, затем вешаем на них обработчик наведения курсора mouseover. Обработчик события наведения мыши в качестве аргумента принимает анонимную функцию, которая срабатывает при наведении курсора.

В этой функции с помощью ключевого слова this мы получаем элемент, на который мы наводим курсор, а затем с помощью функцииanimate мы устанавливаем свойства, которые будут анимироваться при наведении курсора.

Подобным образом работаем второй обработчик mouseout. В итоге при наведении курсора изображение увеличится, а при потере изображением фокуса курсора, оно вернется в первоначальные размеры.

## Введение в AJAX

AJAX (Асинхронный JavaScript и XML) представляет собой технологию гибкого взаимодействия между клиентом и сервером. Благодаря ее использованию мы можем осуществлять асинхронные запросы к серверу без перезагрузки всей страницы. Правда, в настоящее время все больше вместо формата XML используется формат JSON для взаимодействия между клиентом и сервером.

Применительно к ASP.NET MVC использование AJAX вылилось в целую концепцию под названием **"ненавязчивого AJAX"** и **ненавязчивого JavaScript** (*unobtrusive Ajax/JavaScript*). Смысл этой концепции заключается в том, что весь необходимый код JavaScript используется не на самой веб-странице, а помещается в отдельные файлы с расширением *\*.js*. А затем с помощью тега <script> мы а веб-станице ссылаемся на данный файл кода.

Таким образом мы отделяем визуализацию от логики приложения. Что имеет свои плюсы. Так, выделение скрипта в отдельный загружаемый файл увеличивает производительность сайта, поскольку файл сохраняется в кэше и затем от туда подгружается.

### Настройка ненавязчивого JavaScript/ AJAX

Во-первых, соответствующие настройки должны быть указаны в файле Web.config:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <appSettings>    <add key="webpages:Version" value="2.0.0.0" />    <add key="webpages:Enabled" value="false" />    <add key="PreserveLoginUrl" value="true" />    <add key="ClientValidationEnabled" value="true" />    <add key="UnobtrusiveJavaScriptEnabled" value="true" />  </appSettings> |

Во-вторых, нам надо подключить соответствующие файлы JavaScript:

|  |  |
| --- | --- |
| 1  2 | @Scripts.Render("~/scripts/jquery-1.7.1.js")  @Scripts.Render("~/scripts/jquery.unobtrusive-ajax.js") |

Первый файл - общая библиотека jQuery. Второй файл (*jquery.unobtrusive-ajax.js*) подключает к приложению функциональность Ajax-хелперов, например, Ajax-форм. Главное не забыть, подключить основную библиотеку jQuery перед остальными файлами, так как все остальные скрипты, как правило, зависят от нее.

### AJAX-хелперы

Для осуществления ajax-запросов в ASP.NET MVC присутствует такая функциональность, как ajax-хелперы:

|  |  |
| --- | --- |
| **Хелпер** | **Описание** |
| Ajax.ActionLink | Создает гиперссылку на действие контроллера, по нажатию на которую происходит ajax-запрос к этому действию |
| Ajax.RouteLink | Похож на хелпер Ajax.ActionLink, только ссылка создается на определенный маршрут, а не на действие контроллера |
| Ajax.BeginForm | Создает html-форму, которая отправляет ajax-запросы к определенному действию определенного контроллера |
| Ajax.BeginRouteForm | Похож на Ajax.BeginForm, только ajax-запросы направляются не к действию контроллера, к по определенному маршруту |
| Ajax.GlobalizationScript | Создает ссылку на скрипт, который содержит информацию о культуре |
| Ajax.JavaScriptStringEncode | Кодирует строку для использования в JavaScript |

Наиболее часто используются **Ajax.ActionLink** и **Ajax.BeginForm**, поэтому далее мы рассмотрим эти хелперы.

## Ajax-Формы

Итак, после настройки и подключения всех необходимых скриптов мы можем приступить непосредственно к работе с Ajax. Допустим, у нас есть класс Book, содержащий данные о книге, а в БД у нас может находиться несколько книг одного автора. И нам надо реализовать поиск всех книг определенного автора.

Казалось бы, зачем в данном случае Ajax, если мы можем, например, в форму вводить имя автора и отправлять на сервер, а сервер в качестве ответа возвратит нам страницу с нужным результатом. Но, как выше уже говорилось, AJAX поможет нам избежать перезагрузки всей страницы и выполнить загрузку данных в **асинхронном режиме**, что несомненно повышает производительность приложения.

Для начала определим действие контроллера, которое будет отвечать за извлечение из БД нужной информации и передавать извлеченную информацию в частичное представление:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public ActionResult BookSearch(string name)  {      var allbooks = db.Books.Where(a => a.Author.Contains(name)).ToList();      if(allbooks.Count<=0)      {          return HttpNotFound();      }      return PartialView(allbooks);  } |

Итак, действие получает в качестве параметра имя автора и по нему осуществляет поиск в БД. Теперь добавим к представлениям данного контроллера частичное представление *BookSearch.cshtml*, поскольку частичные представления довольно удобны для работы с AJAX:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | @model IEnumerable<AjaxMvcApplication.Models.Book>    <div id="searchresults">        <h3>Все книги автора : @Model.First().Author</h3>      <ul>      @foreach (var item in Model)      {          <li>@item.Name</li>      }      </ul>    </div> |

В данном случае представление типизируется для модели IEnumerable<AjaxMvcApplication.Models.Book>, которая и будет передаваться в представление. А затем в элемент div будут выводиться результаты поиска в виде списка при условии, конечно, если модель не равнаnull.

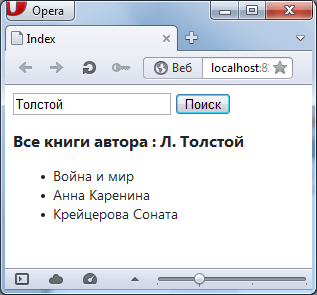
Теперь перейдем к самому представлению, которое и будет отображаться пользователю:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | @{      ViewBag.Title = "Index";  }    <div>  @using (Ajax.BeginForm("BookSearch", new AjaxOptions { UpdateTargetId = "results"}))  {      <input type="text" name="name" />      <input type="submit" value="Поиск" />  }  <div id="results"></div>  </div> |

Хелпер **Ajax.BeginForm** похож на хелпер Html.BeginForm - он также создает элемент form, который используется для отправки запроса на сервер. Первый параметр принимает имя действия, к которому будет обращен запрос. В данном случае это созданное выше действиеBookSearch, которое возвращает частичное представление с данными. Если действие находится не в текущем контроллере, а в другом мы также можем указать имя контроллера: Ajax.BeginForm("BookSearch", "Home", new AjaxOptions....

Второй параметр более интересный. Он представляет объект **AjaxOptions**, который влияет на отображение результатов. Он принимает ряд параметров, из которых мы в данном случае использовали только UpdateTargetId. Этот параметр указывает, что у нас при получении результатов от сервера на странице будет обновляться элемент с id="results", в который помещется html-код выше созданного частичного представления.

Теперь мы можем запустить приложение и осуществить поиск с помощью AJAX:



Обратите внимание на создаваемую разметку для данной формы:

|  |  |
| --- | --- |
| 1  2  3  4  5 | <form action="/Home/BookSearch" data-ajax="true" data-ajax-mode="replace" data-ajax-update="#results" id="form0" method="post">      <input type="text" name="name" />      <input type="submit" value="Поиск" />  </form>  <div id="results"></div> |

Создается как и в случае с хелпером Html.BeginForm элемент form. Но теперь в специальном атрибуте указывается, что это Ajax-форма:data-ajax="true". Остальные параметры формы являются передачей в html параметров объекта AjaxOptions.

## Параметры объекта AjaxOptions

Выше мы использовали только один параметр объекта AjaxOptions. Но он имеет еще ряд параметров, которые позволяют настроить отображение результатов запроса:

* **Confirm** - настраивает сообщение о подтверждении отправки запроса на сервер. Если пользователь не подтвердит, то запрос не будет отправлен
* **HttpMethod** - устанавливает метод (Get или Post), с помощью которого выполняется запрос
* **InsertionMode** - устанавливает, как полученные результаты будут отображаться на странице. Может принимать одно из трех значений перечисления InsertionMode: InsertAfter, InsertBefore и Replace(по умолчанию).
* **LoadingElementId**- устанавливает id элемента html-страницы, который будет отображаться во время запроса. Обычно это какая-нибудь анимация, которая дает знать, что некоторая работа выполняется в фоновом режиме
* **LoadingElementDuration** - устанавливает количество миллисекунд, через которое появится элемент, указанный в параметре LoadingElementId
* **OnBegin** - задает обратный вызов перед отправкой запроса. Соотносится с событием beforeSend библиотеки jQuery
* **OnComplete** - задает обратный вызов, который вызывается после выполнения запроса (как удачного, так и неудачного). Соотносится с событием complete библиотеки jQuery
* **OnFailure** - задает обратный вызов, который вызывается после неудачного выполнения запроса. Соотносится с событием errorбиблиотеки jQuery
* **OnSuccess** - задает обратный вызов, который вызывается после удачного выполнения запроса. Соотносится с событием successбиблиотеки jQuery
* **UpdateTargetId** - указывает на id элемента, в котором будут выводиться результаты запроса
* **Url** - устанавливает адрес Url сервера, на который отправляется запрос. Установив данное свойство, можно не использовать название имя контроллера и его действие в качестве параметров Ajax.BeginForm

### LoadingElementId и LoadingElementDuration

Эти параметры используются для создания некой анимации или визуализации, которая извещает пользователей о процессе запроса. Если запрос идет долго, то пользователь может долго не получать результаты запроса и может подумать, что страница зависла. Чтобы пользователь видел, что процесс выполняется, и нужно настраивать данные параметры.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | @using (Ajax.BeginForm("BookSearch", new AjaxOptions { UpdateTargetId = "results", LoadingElementId="loading",                                                          LoadingElementDuration=1000}))  {      <input type="text" name="name" />      <input type="submit" value="Поиск" />  }  <div id="loading" style="display:none; color:Red; font-weight:bold;">      <p>Идет загрузка...</p>  </div>  <div id="results"></div> |

Данный элемент отображается на странице только во время запроса, поэтому для него устанавливается атрибут display:none. И после выполнения запроса данный элемент снова скрывается.

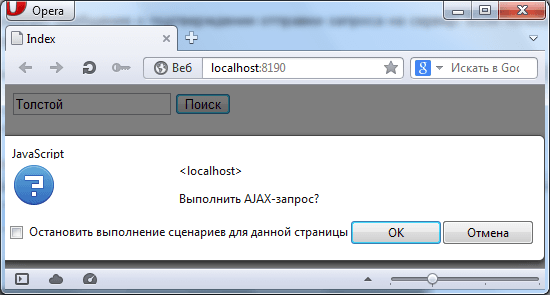
Очень популярен способ индикации процесса с помощью картинки:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @using (Ajax.BeginForm("BookSearch", new AjaxOptions { UpdateTargetId = "results", LoadingElementId="loading",                                                          LoadingElementDuration=1000}))  {      <input type="text" name="name" />      <input type="submit" value="Поиск" />  }  <img id="loading" src="@Url.Content("~/Content/Images/loader.gif")" style="display:none" />  <div id="results"></div> |

### Confirm

Этот параметр настраивает сообщение о подтверждении отправки запроса на сервер. Если пользователь не подтвердит, то запрос не будет отправлен.

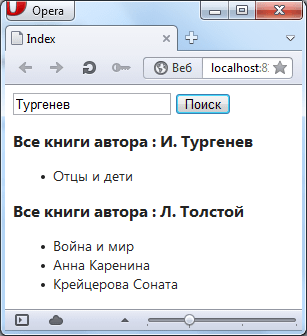
|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @using (Ajax.BeginForm("BookSearch", new AjaxOptions { UpdateTargetId = "results", Confirm="Выполнить AJAX-запрос?"}))  {      <input type="text" name="name" />      <input type="submit" value="Поиск" />  }  <div id="results"></div> |



### InsertionMode

Этот параметр указывает на способ вставки результатов запроса на страницу. Он принимает одно из трех значений одноименного перечисления InsertionMode: InsertAfter (после остальных результатов), InsertBefore (перед остальными результатами) и Replace (полность замещает результаты предыдущего AJAX-запроса - используется по умолчанию).

Так, при установке значения параметра InsertionMode=InsertionMode.InsertBefore мы будем наблюдать следующую картину:



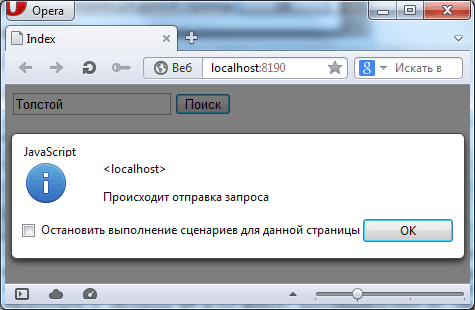
### Обратные вызовы

Параметры OnBegin, OnComplete, OnFailure и OnSuccess помогают задать обратные вызовы, которые будут выполняться перед запросом, либо в случае удачного или неудачного выполнения запроса.

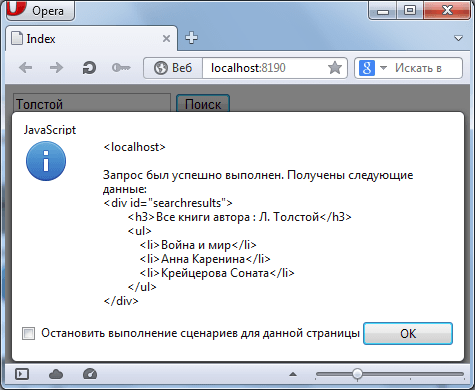
Чтобы их использовать, определим соответствующие обработчики javascipt в представлении и укажем их в параметрах объекта AjaxOptions:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | <script type="text/javascript">      function OnBegin() {          alert("Происходит отправка запроса");      }      function OnSuccess(data) {          alert("Запрос был успешно выполнен. Получены следующие данные: \n" + data);      }      function OnFailure(request, error) {          $("#results").html("Книги указанного автора не содержатся в базе данных.");      }      function OnComplete(request, status) {          alert("Статус запроса : " + status);      }  </script>    @using (Ajax.BeginForm("BookSearch", new AjaxOptions { UpdateTargetId = "results", OnBegin="OnBegin",                                              OnFailure="OnFailure", OnSuccess="OnSuccess", OnComplete="OnComplete"}))  {      <input type="text" name="name" />      <input type="submit" value="Поиск" />  }    <div id="results"></div> |

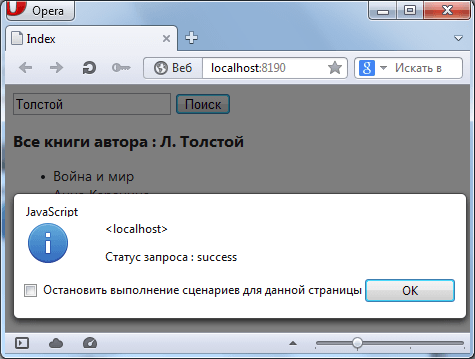
##### Обратный вызов OnBegin



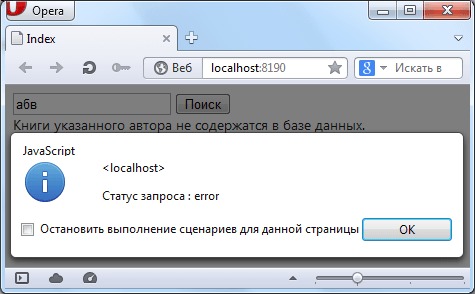
##### Обратный вызов OnSuccess



##### Обратный вызов OnComplete



##### Обратный вызов OnFailure



## AJAX-ссылки

Другим часто используемым AJAX-хелпером является **Ajax.ActionLink**. Он во многом похож на хелпер Ajax.BeginForm за тем исключением, что генерирует специальные ajax-ссылки. Посмотрим на примере. Добавим в представление следующую разметку:

|  |  |
| --- | --- |
| 1  2  3 | <div id="bestbook">      @Ajax.ActionLink("Лучшая книга", "BestBook", new AjaxOptions{ UpdateTargetId="bestbook"})  </div> |

В данном случае мы также используем объект AjaxOptions для установки параметров асинхронной ссылки, который принимает все те же параметры, что и в случае с ajax-формой.

Мы помещаем ссылку в элемент с id, который указывается в параметре UpdateTargetId. Таким образом, результат запроса будет замещать ссылку. Но для работы нам также необходимо, как и в случае с ajax-формой, действие контроллера и вызываемое им частичное представление.

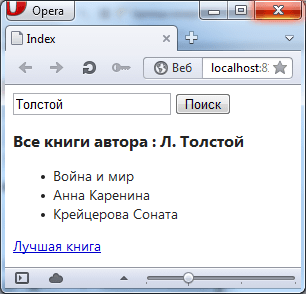
Метод контроллера, в данном случае BestBook, будет просто возвращать первую книгу:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public ActionResult BestBook()  {      Book book = db.Books.First();      return PartialView(book);  } |

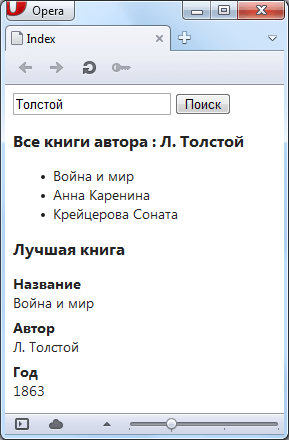
А вызываемое им частичное представление *BestBook.cshtml* будет выводить информацию об этой книге:

|  |  |
| --- | --- |
| 1  2  3 | @model AjaxMvcApplication.Models.Book  <h3>Лучшая книга</h3>  @Html.DisplayForModel() |

AJAX-ссылка будет отображена при запуске как обычная ссылка:



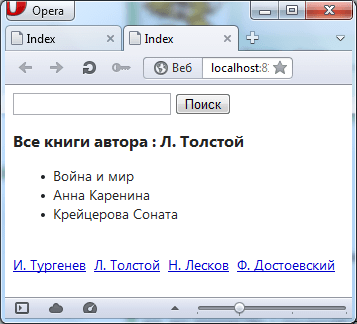
А после нажатия она будет заменена полученным от сервера контентом:



В качестве примера можно также изменить код представления из предыдущего раздела про ajax-формы, применив ajax-ссылки:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | @using (Ajax.BeginForm("BookSearch", new AjaxOptions { UpdateTargetId = "results"}))  {      <input type="text" name="name" />      <input type="submit" value="Поиск" />  }    <div id="results"></div>  <br />  @foreach(string s in ViewBag.Authors)  {      <div style="margin-right:8px;float:left;">          @Ajax.ActionLink(s, "BookSearch", new {name=s},new AjaxOptions{ UpdateTargetId="results"})      </div>  } |

В итоге все авторы, передаваемые в представление через ViewBag, выводятся в виде ссылок внизу страницы. А нажатие на одну из этих ссылок будет иметь то же действие, что и поиск по кнопке:



## Формат JSON

Как уже писалось в начале главы, технология AJAX все больше вместо передачи данных в формате xml использует формат JSON (JavaScript Object Notation). JSON не зависит от языка программирования, он более удобен и легче обрабатывается. Например, объект Book в формате JSON мог бы выглядеть бы так: {"Name":"Война и мир", "Author":"Л. Толстой", "Year":"1863"}.

В JSON каждый отдельный объект заключается в фигурные скобки и представляет собой набор пар ключ-значение, разделенных запятыми, где ключом является название свойства объекта, а значением соответственно значение этого свойства.

Ранее мы в ajax-форме обращались к методу, который возвращал данные в виде обычной разметки html. Теперь изменим его или определим новый, чтобы он возвращал данные в формате JSON:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public JsonResult JsonSearch(string name)  {      var jsondata = db.Books.Where(a => a.Author.Contains(name)).ToList<Book>();      return Json(jsondata, JsonRequestBehavior.AllowGet);  } |

Новое действие теперь возвращает объект **JsonResult**, который принимает объект с результатами запроса (в данном случае объект jsondata). Второй необязательный параметр представляет значение перечисления JsonRequestBehavior и может принимать два значения:AllowGet (разрешить Get-запросы) и DenyGet (запретить Get-запросы). В данном случае мы разрешаем действию посылать результаты в JSON-формате в ответ на запросы Get.

Теперь изменим ajax-форму в нашем представлении:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | <script type="text/javascript">      function OnSuccess(data) {          var results = $('#results'); // получаем нужный элемент          results.empty(); //очищаем элемент          for (var i = 0; i < data.length; i++) {              results.append('<li>' + data[i].Name + '</li>'); // добавляем данные в список          }      }  </script>    @using (Ajax.BeginForm("JsonSearch", new AjaxOptions {OnSuccess = "OnSuccess"}))  {      <input type="text" name="name" />      <input type="submit" value="Поиск" />  }  <br />  <div id="results"></div> |

Поскольку параметр UpdateTargetId не будет приниматься во внимание при получении данных в формате JSON, поэтому вся логика по выводу данных на страницу закладывается в обратный вызов OnSuccess. В обработчике OnSuccess мы последовательно добавляем полученные данные в нужный элемент html-страницы.

## AJAX-запросы с помощью jQuery

Кроме использования AJAX-форм и AJAX-ссылок мы можем выполнить запрос средствами самой библиотеки jQuery, не прибегая к ajax-хелперам.

Для примера создадим представление, в котором и будут осуществляться запросы AJAX:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | <input type="text" name="name" id="search" />  <input type='button' id="submit" value='Поиск' />  <div id="results"></div>  <script type="text/javascript">      $(document).ready(function () {          $('#submit').click(function (e) {              e.preventDefault();              var name = $('#search').val();              name = encodeURIComponent(name);              $('#results').load("<http://localhost:51805/Home/BookSearch?name=>" + name);          });      });  </script> |

В данном случае мы в функции jQuery вешаем на элемент с id=submit (а это кнопка) обработчик события click. Строкой var name = $('#search').val(); мы получаем введенное в текстовое поле значение. Следующей строчкой обрабатываем его - кодируем пробельные символы, так как иначе, если введенное слово будет содержать пробельные символы, то передаваемое в действие контроллера значение будет некорректно.

И в строке $('#results').load("http://localhost:51805/Home/BookSearch?name=" + name); мы загружаем в элемент с id=results контент, полученный с помощью метода load.

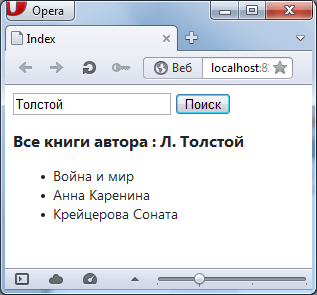
Метод, к которому мы обращаемся, будет получать строковый параметр, осуществлять по нему поиск и передавать найденные данные в частичное представление:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public ActionResult BookSearch(string name)  {      List<Book> books = db.Books.Where(a => a.Author.Contains(name)).ToList<Book>();      return PartialView(books);  } |

Вызываемое частичное представление могло бы выглядеть так:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | @model IEnumerable<AjaxMvcApplication.Models.Book>    <div id="searchresults">      @if (Model != null  && Model.Count()>0)      {          <h3>Все книги автора : @Model.First().Author</h3>          <ul>          @foreach (var item in Model)          {              <li>@item.Name</li>          }          </ul>      }  </div> |

Теперь, мы можем осуществить поиск, введя какое-либо значение и нажав на кнопку поиска:



Выше был представлен только один способ проведения ajax-запросов с помощью jquery. Более подробно об ajax-запросах и jquery вы можете прочитать здесь: [Технология Ajax](http://metanit.com/web/jquery/6.1.php)

## Работа с jQuery UI

Одним из наиболее популярных плагинов jQuery является jQuery UI, поэтому его и включили в стандартный набор скриптов. Этот плагин предназначен для работы с пользовательским интерфейсом и содержит, во-первых, ряд интересных визуальных эффектов, типа bounce, explode, fade, pulsate и shake. Во-вторых, он содержит набор виджетов, как accordion, autocomplete, button, datepicker, dialog, progressbar, slider и tabs.

Чтобы начать работать с jQuery UI, надо прежде всего подключить эту библиотеку:

|  |  |
| --- | --- |
| 1  2 | <link href="@Url.Content("~/Content/themes/base/jquery-ui.css")" rel="stylesheet" type="text/css" />  <script src="@Url.Content("~/Scripts/jquery-ui-1.8.20.min.js")" type="text/javascript"></script> |

Обратите внимание, что в данном случае нам также нужно подключить и файл стилей *jquery-ui.css*, иначе плагин просто не будет работать. Дело в том, что для отображения и визуализации плагин использует темы. В данном случае в проектах по типу Basic и Internet Application по умолчанию включена одна тема - base. Все используемые ее файлы располагаются в папке *Content/themes/base*. Здесь есть изображения, нужные для визуализации. А также файлы стилей. Файл *jquery-ui.css* является основным, поэтому мы его подключаем к приложению.

### Виджет Autocomplete

Теперь добавим в наше приложение виджет Autocomplete, который реализует функциональность автозаполнения.

Для начала определим в представлении элемент ввода, для которого и будет производиться автозаполнение:

|  |  |
| --- | --- |
| 1  2 | <input type="text" name="name" data-autocomplete-source="@Url.Action("AutocompleteSearch", "Home")" />  <input type="submit" value="Поиск" /> |

Итак, чтобы подключить автозаполнение, мы добавляем к элементу ввода атрибут HTML5 data-autocomplete-source. Этот атрибут указывает на источник данных. В данном случае источником данных будет действие AutocompleteSearch в контроллере Home:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public ActionResult AutocompleteSearch(string term)  {      var authors = db.Books.Where(a => a.Author.Contains(term)).ToList().Select(a => new { value = a.Author }).Distinct();        return Json(authors, JsonRequestBehavior.AllowGet);  } |

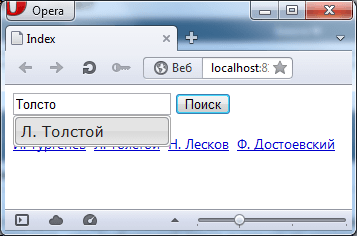
Мы просто извлекаем все книги из базы, по ним формируем новый анонимный объект - автора, и получившийся массив новых объектов посылаем обратно клиенту в формате JSON. Причем каждый из создаваемых анонимных объектов должен иметь свойство label или свойство value (как в нашем случае), либо оба этих свойства. Свойство label используется при отображении текста пользователю. При выборе пользователем определенного элемента в списке объектов виджет помещает value выделенного элемента в элемент ввода на странице. Если либо свойство label, либо value не указано, виджет будет использовать значение одного указанного свойства как для value, так и для label.

И в конце надо добавить скрипт, поместив его где-нибудь представлении или во внешний js-файл, который вызывает функцию autocomplete для элемента заполнения:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | $(function () {          $("[data-autocomplete-source]").each(function () {              var target = $(this);              target.autocomplete({ source: target.attr("data-autocomplete-source") });          });      }); |

Функция each вызывает для каждого найденного по селектору элемента функцию, указанную в параметре. В этой функции вызывается метод плагина autocomplete для каждого элемента. Параметр, передаваемый в метод autocomplete, представляет собой объект, в котором определено одно свойство - свойство source. Но можно установить и большее количество свойств, например, задержку после нажатия клавиши или минимальное количество символов, необходимое для срабатывания автозаполнения и отправки запрос к источнику данных.

В итоге при обращении к источнику данных (в данном случае - к действию AutocompleteSearch контроллера Home) плагин передает текущее значение элемента ввода в качестве параметра term в строке запроса, а обратно клиенту отсылается массив объектов в формате JSON, которые на клиенте преобразуются в форму для отображения:



# Авторизация и аутентификация в MVC 4

## Введение в авторизацию и аутентификацию

Большую роль в веб-приложениях играют механизмы авторизации и аутентификации. Они позволяют разграничить доступ для различных групп пользователей, а также идентифицировать пользователей.

Аутентификация - это процесс идентицификации пользователя, то есть грубо говоря мы узнаем, что за пользователь посетил веб-приложение. А авторизация уже представляет процесс определения прав, которые могут быть даны аутентифицированному пользователю, его возможностей по доступу к ресурсам веб-приложения.

В главе, посвященной фильтрам, мы уже разбирали один момент, связанный с авторизацией - фильтр авторизации. Он используется в виде атрибута Authorize, который устанавливается либо для одиночного действия контроллера, либо для всего контроллера.

|  |  |
| --- | --- |
| 1  2  3  4  5 | [Authorize]  public ActionResult Index()  {      return View();  } |

И после установки атрибута к данному ресурсу получат доступ только те пользователи, которые успешно осуществили вход в веб-приложение.

Мы также можем разрешить неавторизованным пользователям доступ к ресурсам, пометив их атрибутом AllowAnonymous.

Авторизация в MVC опирается на три ключевых области, которые помогают управлять пользователями в системе. Это членство (membership), роли и профили.

Членство представляет управление регистрационными данными пользователей, которые затем используются для авторизации и аутентификации.

Роли - группы пользователей, которые используются для авторизации и разделения доступа к ресурсам.

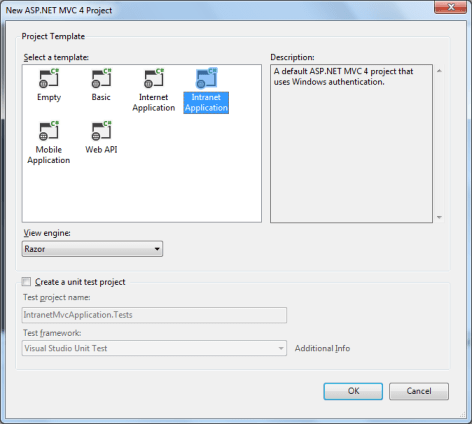
Профили представляют различную информацию о пользователях, как например, имя, фамилию, рост, возраст и т.д.

Управление пользователями, членством и ролям производится с помощью провайдеров членства и ролей. В MVC 4 можно использовать различные провайдеры: как в виде API SimpleMembershipProvider, так и в виде универсальных провайдеров. При необходимости гибкая среда ASP.NET MVC позволяет создавать и использовать свои провайдеры членства/ролей и профилей со своей уникальной логикой.

## Аутентификация Windows

Аутентификация Windows представляет один из способов аутентификации пользователя в системе. При использовании этой модели аутентификации при обращении пользователя к ресурсам веб-приложения вместе с HTTP-запросом посылается и токен безопасности Windows, который и верифицирует пользователя.

Аутентификацию на основе учетных записей Windows наиболее удобно применять к веб-приложениям, создаваемым по шаблону **Intranet Application**. Создадим такое приложение:



По сути Intranet-приложение похоже на шаблон Internet Application, поскольку имеет уже готовую инфраструктуру. Однако оно нацелено именно на использование аутентификации Windows, поэтому мы не увидим в отличие от Internet Application контроллера AccountContoller и моделей, представляющих учетные записи.

Аутентификация Windows задана следующей строкой в файле *web.config*:

|  |  |
| --- | --- |
| 1 | <authentication mode="Windows" /> |

Теперь настроим приложение на данный способ аутентификации. Для этого перейдем к свойствам проекта и установим для свойства**Anonymous Authentication** значение **Disabled**, а для свойства **Windows Authentication** значение **Enabled**:

Теперь вся основная настройка сделана, и мы можем войти в приложение. При запуске нам будет предложено ввести логин и пароль. Поскольку мой логин на локальному компьютере "Eugene", то я и ввожу соответствующий логин, а также пароль, под которым я вхожу на своем локальном компьютере:

В случае, если все данные введены верно, то пользователь успешно войдет в систему:

### Настройка авторизации

Применяя фильтры авторизации, мы можем ограничить доступ к действиям контроллера или всему контроллеру для определенных ролей (используя группы в Windows) или пользователей:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | [Authorize (Users=@"Eugene-ПК\Eugene")]  public ActionResult Index()  {      return View();  }  [Authorize(Roles="Admin, AllUsers")]  public ActionResult About()  {      return View();  }  [Authorize(Roles = "Admin")]  public ActionResult Contact()  {      return View();  } |

## Аутентификация форм

Еще один способов аутентификации пользователей представляет аутентификация форм. Она более гибкая по сравнению с аутентификацией Windows, хотя, возможно, и чуть более сложная в понимании. Она основывается она выдаче аутентифицированному пользователю куки-наборов, по которым он в дальнейшем верифицируется.

Чтобы установить режим аутентификации на основе форм, в файл конфигурации web.config заносится следующие строки в секцию*system.web*:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <system.web>    ..................    <authentication mode="Forms">      <forms loginUrl="~/Account/Login" timeout="2880" />    </authentication>    ............  </system.web> |

Блок *authentication* определяет параметры аутентификации, в частности, блок forms задает в параметре **loginUrl** ресурс, на который будет направлен пользователь, который не был аутентифицирован.

Для рассмотрения аутентификации форм возьмем веб-приложение по шаблону Internet Application, который содержит всю базовую инфраструктуру авторизации пользователей.

Это приложение в файле конфигурации по умолчанию содержит подключение к бд. При желании мы можем сами создать свою БД и переопределить подключение. А база данных для этого подключения, создаваемая при первом обращении пользователя в папке App\_Data, хранит все данные об учетных записях, зарегистрированных в системе. Из этой же базы будут браться данные для подтверждения аутентификации пользователя.

Вся логика аутентификации заключена в методах Login контроллера AccountController, которые осуществляют вход пользователя в систему. В зависимости от используемой версии Visual Studio. Так, в Visual Studio 2010 метод Login выглядит следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | [AllowAnonymous]  public ActionResult Login(string returnUrl)  {      ViewBag.ReturnUrl = returnUrl;      return View();  }    [AllowAnonymous]  [HttpPost]  public ActionResult Login(LoginModel model, string returnUrl)  {      if (ModelState.IsValid)      {          if (Membership.ValidateUser(model.UserName, model.Password))          {              FormsAuthentication.SetAuthCookie(model.UserName, model.RememberMe);              if (Url.IsLocalUrl(returnUrl))              {                  return Redirect(returnUrl);              }              else              {                  return RedirectToAction("Index", "Home");              }          }          else          {              ModelState.AddModelError("", "The user name or password provided is incorrect.");          }      }      return View(model);  } |

Тогда как в проекте, созданном в Visual Studio 2012 этот метод будет выглядеть следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | [AllowAnonymous]  public ActionResult Login(string returnUrl)  {      ViewBag.ReturnUrl = returnUrl;      return View();  }    [HttpPost]  [AllowAnonymous]  [ValidateAntiForgeryToken]  public ActionResult Login(LoginModel model, string returnUrl)  {      if (ModelState.IsValid && WebSecurity.Login(model.UserName, model.Password,                                  persistCookie: model.RememberMe))      {          return RedirectToLocal(returnUrl);      }        ModelState.AddModelError("", "The user name or password provided is incorrect.");      return View(model);  } |

Несмотря на некоторые различия фактически они производят те же действия в виде аутентификации форм. Сама аутентификация разделяется на два действия:

* **Валидация пользователя**: в первом случае это делает метод Membership.ValidateUser. И если пользователь находится в нашей базе данных, то далее приложение переходит ко второму этапу.
* Второй этап представляет **создание аутентификационного тикета** - то есть некоторого билета безопасности, по которому веб-приложение будет опознавать пользователя. Этот тикет приложение устанавливает для браузера в виде куки-набора по имени**.AUTHPATH** с помощью метода FormsAuthentication.SetAuthCookie

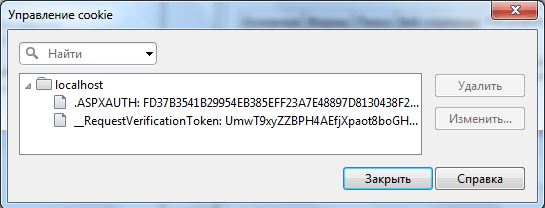
В этом и по сути и состоит сущность аутентификации форм.

Несмотря на то, что во втором случае - в коде для Visual Studio 2012 не используются явно эти методы, а вместо них применяется только один - WebSecurity.Login, но фактически он скрывает ту же функциональность в своей внутренней реализации:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public static bool Login(string userName, string password, bool persistCookie = false)  {      VerifyProvider();      bool success = Membership.ValidateUser(userName, password);      if (success)      {          FormsAuthentication.SetAuthCookie(userName, persistCookie);      }      return success;  } |

### Создание куки-набора

Важно отметить, что при установке куки-наборов их содержимое шифруется при помощи машинных ключей, которую автоматически создает IIS на сервере. После успешной авторизации на сервере приложения мы можем посмотреть в браузере добавленный куки-набор:



Как вы видите, первый куки-набор **.AUTHPATH**, который содержит данные авторизованного пользователя, совершенно нечитаем, что уменьшает вероятность взломов простой подделкой занчений куки.

Второй куки-набор - **\_\_RequestVerificationToken** - представляет специальный куки-набор, который препятствует взлому в виде CSRF-атак.

### Настройка аутентификации

Используя атрибуты узла *forms* в файле конфигурации, мы можем настроить параметры аутентификации. В частности мы можем применить следующие параметры:

* **coockieless**: определяет, применяются ли куки-наборы и как они используются. Может принимать следующие значения:**UseCookies**(определяет, что куки-наборы будут использоваться всегда вне зависимости от устройства), **UseUri** (куки-наборы никогда не используются), **AutoDetect** (если устройство поддерживает куки-наборы, то они используются, в противном случае они не применяются, при этом проводится тестирование, определяющее, включена ли поддержка), **UseDeviceProfile** (используется по умолчанию) (если устройство поддерживает куки-наборы, то они используются, в противном случае они не применяются, в отличие от предыдущего случая тестирование не проводится)
* **defaultUrl**: определяет путь, по которому осуществляется переход после авторизации
* **domain**: определяет куки-наборы для всего домена. Благодаря этому мы можем использовать одни и те же куки-набры для главного домена и его субдоменов. По умолчанию имеет значение в качестве пустой строки
* **loginUrl**: адрес для аутентификации пользователя. Значение по умолчанию - "~/Account/Login"
* **name**: задает имя для куки-набора. Значение по умолчанию - ".ASPXAUTH"
* **path**: задает путь для куки-наборов. Значение по умолчанию - "/"
* **requireSSL**: определяет, требуется ли SSL-соединение для передачи куки-наборов. Значение по умолчанию false
* **timeout**: определяет срок действия куков в минутах

Например, мы можем задать следующее определение аутентификации форм:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <system.web>    ..................    <authentication mode="Forms">      <forms loginUrl="~/Account/Login" timeout="600" name="cookies" />    </authentication>    ............  </system.web> |

## Авторизация в MVC 4 и SimpleMembershipProvider

В ASP.NET MVC 4 был введен новый провайдер членства, который стал использоваться по умолчанию в проектах по типу Internet Application - SimpleMembershipProvider. Класс SimpleMembershipProvider призван упростить работу с пользователями и ролями, обладая большей гибкостью и расширяемостью по сравнению с традиционным провайдером членства. Представляемое этим классом API по сути применяет те же провайдеры, которые содержатся в основе ASP.NET API. Так, класс **SimpleRoleProvider** просто применяет абстрактный базовый класс RoleProvider.

Класс SimpleMembershipProvider также применяет традиционный провайдер членства: сначала абстрактный класс ExtendedMembershipProvider наследует MembershipProvider, а затем сам SimpleMembershipProvider наследуется от ExtendedMembershipProvider.

Для облегчения работы с новым провайдером членства используется класс-хелпер **WebSecurity** из пространства имен WebMatrix.WebData. Через этот класс мы можем инициализировать подключение или проводить верифицикацию пользователей.

В качестве примера для рассмотрения выберем проект по типу Internet Application. После создания проекта в каталоге Filters в структуре проекта мы можем найти файл **InitializeSimpleMembershipAttribute.cs**, который содержит объявление класса**InitializeSimpleMembershipAttribute**. Этот класс является производным от ActionFilterAttribute, то есть фактически представляет класс фильтра действия, применяемого к контроллеру AccountController.

InitializeSimpleMembershipAttribute содержит определение класса-инициализатора SimpleMembershipProvider, который создает соединение с базой данных следующим вызовом:

|  |  |
| --- | --- |
| 1 | WebSecurity.InitializeDatabaseConnection("DefaultConnection", "UserProfile", "UserId", "UserName", autoCreateTables: true); |

По умолчанию этот метод определяет следующие значения:

* "DefaultConnection" - имя подключение, которое содержится в файле Web.config
* "UserProfile" - название таблицы пользователей в БД (по умолчанию содержит столбцы UserId и UserName). Именно эта таблица хранит профили пользователей.
* "UserId" - имя столбца в таблице пользователей, который содержит Id пользователя
* "UserName" - имя столбца в таблице пользователей, который содержит имя пользователя
* autoCreateTables: true - создает весь необходимый для провайдера Simple Membership набор таблиц, если они еще не существуют.

При первом обращении к БД, если ее не существовало, она будет создана и автоматически заполнена таблицами с определенными полями.

Эти таблицы имеют следующее определение:

#### webpages\_Membership

|  |  |
| --- | --- |
| **Столбец** | **Тип данных** |
| UserId | int |
| CreateDate | datetime |
| ConfirmationToken | nvarchar(128) |
| IsConfirmed | bit |
| LastPasswordFailureDate | datetime |
| PasswordFailuresSinceLastSuccess | int |
| Password | nvarchar(128) |
| PasswordChangedDate | datetime |
| PasswordSalt | nvarchar(128) |
| PasswordVerificationToken | nvarchar(128) |
| PasswordVerificationTokenExpirationDate | datetime |

Таблица webpages\_Membership используется для управления учетными записями пользователей, то есть членством.

#### webpages\_OAuthMembership

|  |  |
| --- | --- |
| **Столбец** | **Тип данных** |
| Provider | nvarchar(30) |
| ProviderUserId | nvarchar(100) |
| UserId | int |

#### webpages\_Roles

|  |  |
| --- | --- |
| **Столбец** | **Тип данных** |
| RoleId | int |
| RoleName | nvarchar(256) |

#### webpages\_UsersInRoles

|  |  |
| --- | --- |
| **Столбец** | **Тип данных** |
| UserId | int |
| RoleId | int |

Последние две таблицы используются для управления ролями и пользователей.

Почему создаются таблицы именно с этими именами? Эти имена определены в коде самого SimpleMembershipProvider:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | //..................................  internal static string MembershipTableName  {      get { return "webpages\_Membership"; }  }    internal static string OAuthMembershipTableName  {      get { return "webpages\_OAuthMembership"; }  }  //........................ |

Другие две таблицы закодированы в коде SimpleRoleProvider:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | internal static string RoleTableName  {      get { return "webpages\_Roles"; }  }    internal static string UsersInRoleTableName  {      get { return "webpages\_UsersInRoles"; }  } |

Чтобы сменить способ организации данных в таблицах нам придется создать собственные провайдеры членства и ролей. (Просмотреть код провайдеров можно по следующим ссылкам: [SimpleRoleProvider](http://aspnetwebstack.codeplex.com/SourceControl/changeset/view/e0115a823029#src/WebMatrix.WebData/SimpleRoleProvider.cs) и [SimpleMembershipProvider](http://aspnetwebstack.codeplex.com/SourceControl/changeset/view/e0115a823029#src/WebMatrix.WebData/SimpleMembershipProvider.cs)

Однако таблицу определения пользователей, которая по умолчанию называется *UserProfile*, мы вполне можем переопределить. В следующем разделе мы это попробуем сделать.

По умолчанию регистрация пользователя в методе Register контроллера AccountController проводится с помощью следующего кода:WebSecurity.CreateUserAndAccount(model.UserName, model.Password). Передавая в метод введенные данные, мы создаем две записи - одну запись в таблице пользователей UserProfiles, где создается запись о пользователе, и связанная с ней запись в таблице членства webpages\_Membership, где хранится пароль и другие учетные данные пользователя. SimpleMembershipProvider использует для шифрования пароля алгоритм PBKDF2, после шифрования зашифрованный пароль попадает в таблицу webpages\_Membership в соответствующее поле.

Затем уже по логину и паролю мы можем войти на сайт - за это отвечает строка WebSecurity.Login(model.UserName, model.Password) в методе Login контроллера AccountController. Здесь проверяются на валидность введенный логин и пароль, и если они проходит проверку, то пользователь аутентифицируется.

SimpleMembershipProvider, будучи довольно гибким классом, может работать с различными типами SQL Server: SQL Server, SQL Azure, SQL Server CE, SQL Server Express, LocalDB. В тоже время он имеет ограничения: так он не может работать с СУБД MySQL, базами данных NoSQL, поскольку использует специфичные для MS SQL Serverа выражения SQL для управления данными.

## Настройка использования SimpleMembershipProvider

SimpleMembershipProvider является довольно гибким классом и его использовать довольно легко. Так мы можем переопределить таблицу пользователей, которая по умолчанию называется *UserProfile* и ее определение. Например, мы можем создать в базе данных таблицу и использовать ее в качестве хранилища данных пользователей, потому что стандартная таблица хранит только логины и id пользователей, а нам захотелось ее расширить: например, чтобы пользователь в качестве логина использовал email, а также добавить в таблицу ряд дополнительных столбцов.

Поэтому добавим в папку *App\_Data* новую базу данных, назовем ее к примеру StoreDB и соответственно изменим строку подключения к этой бд в файле web.config, чтобы бд использовалась по умолчанию:

|  |  |
| --- | --- |
| 1  2  3 | <connectionStrings>      <add name="DefaultConnection" connectionString="Data Source=(LocalDB)\v11.0;AttachDbFilename='|DataDirectory|\StoreDB.mdf';Integrated Security=True" providerName="System.Data.SqlClient" />  </connectionStrings> |

Добавим в базу данных таблицу *Users* и создадим для нее следующее определение полей:

Далее перейдем в файл *InitializeSimpleMembershipAttribute.cs* и изменим строку инициализации подключения к БД в соответствии с определением нашей таблицы

|  |  |
| --- | --- |
| 1 | WebSecurity.InitializeDatabaseConnection("DefaultConnection", "Users", "Id", "Email", autoCreateTables: true); |

Поскольку наша таблица называется Users, идентификатор пользователя содержится в столбце Id, а его логин в столбце Email.

Теперь нам надо изменить используемые по умолчанию модели, чтобы они соответствовали определению столбцов таблицы пользователей. Поэтому изменим следующие классы в файле *AccoutModels.cs*. Класс контекста:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public class UsersContext : DbContext  {      public UsersContext()          : base("DefaultConnection")      {      }        public DbSet<User> Users { get; set; }  } |

Класс пользователей User, который содержит данные об отдельном пользователе:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | [Table("Users")]  public class User  {      [Key]      [DatabaseGeneratedAttribute(DatabaseGeneratedOption.Identity)]      public int Id { get; set; }      public string Email { get; set; }      public string FirstName { get; set; }      public string LastName { get; set; }      public int Year { get; set; }  } |

Классы моделей регистрации и логина:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | public class LoginModel  {      [Required]      [Display(Name = "Адрес электронной почты")]      public string Email { get; set; }        [Required]      [DataType(DataType.Password)]      [Display(Name = "Пароль")]      public string Password { get; set; }        [Display(Name = "Запомнить")]      public bool RememberMe { get; set; }  }    public class RegisterModel  {      [Required]      [DataType(DataType.EmailAddress)]      [Display(Name = "Адрес электронной почты")]      public string Email { get; set; }        [Required]      [StringLength(100, ErrorMessage = "Пароль не может быть меньше 6 и больше 100 символов", MinimumLength = 6)]      [DataType(DataType.Password)]      [Display(Name = "Password")]      public string Password { get; set; }        [DataType(DataType.Password)]      [Display(Name = "Confirm password")]      [Compare("Password", ErrorMessage = "пароли не совпадают")]      public string ConfirmPassword { get; set; }        [Required]      [Display(Name = "Имя")]      public string FirstName { get; set; }        [Required]      [Display(Name = "Фамилия")]      public string LastName { get; set; }        [Required]      [Display(Name = "Год рождения")]      public int Year { get; set; }  } |

Теперь изменим код контроллера AccountController, чтобы он регистрировал нового пользователя должным образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48 | [AllowAnonymous]  public ActionResult Login(string returnUrl)  {      ViewBag.ReturnUrl = returnUrl;      return View();  }    [HttpPost]  [AllowAnonymous]  [ValidateAntiForgeryToken]  public ActionResult Login(LoginModel model, string returnUrl)  {      if (ModelState.IsValid && WebSecurity.Login(model.Email, model.Password, persistCookie: model.RememberMe))      {          return RedirectToLocal(returnUrl);      }        ModelState.AddModelError("", "Неверный пароль или логин");      return View(model);  }    [AllowAnonymous]  public ActionResult Register()  {      return View();  }    [HttpPost]  [AllowAnonymous]  [ValidateAntiForgeryToken]  public ActionResult Register(RegisterModel model)  {      if (ModelState.IsValid)      {          try          {              WebSecurity.CreateUserAndAccount(model.Email, model.Password,                  new { FirstName=model.FirstName, LastName=model.LastName, Year=model.Year});              WebSecurity.Login(model.Email, model.Password);              return RedirectToAction("Index", "Home");          }          catch (MembershipCreateUserException e)          {              ModelState.AddModelError("", ErrorCodeToString(e.StatusCode));          }      }      return View(model);  } |

Если в случае с действием Login мы по сути просто меняем имя свойства, которое определяет логин, то в действии Register мы также добавляем в БД переданные данные о пользователе через анонимный объект: WebSecurity.CreateUserAndAccount(model.Email, model.Password, new { FirstName=model.FirstName, LastName=model.LastName, Year=model.Year});

И последнее, что осталось изменить - это представления логина и регистрации. Представление логина:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | @model AuthInternetApplication.Models.LoginModel    @{      ViewBag.Title = "Логин";  }    <hgroup class="title">      <h1>@ViewBag.Title.</h1>  </hgroup>    <section id="loginForm">  <h2>Вход с помочью учетной записи</h2>  @using (Html.BeginForm(new { ReturnUrl = ViewBag.ReturnUrl })) {      @Html.AntiForgeryToken()      @Html.ValidationSummary(true)        <fieldset>          <ol>              <li>                  @Html.LabelFor(m => m.Email)                  @Html.TextBoxFor(m => m.Email)                  @Html.ValidationMessageFor(m => m.Email)              </li>              <li>                  @Html.LabelFor(m => m.Password)                  @Html.PasswordFor(m => m.Password)                  @Html.ValidationMessageFor(m => m.Password)              </li>              <li>                  @Html.CheckBoxFor(m => m.RememberMe)                  @Html.LabelFor(m => m.RememberMe, new { @class = "checkbox" })              </li>          </ol>          <input type="submit" value="Войти" />      </fieldset>      <p>          @Html.ActionLink("Регистрация", "Register")      </p>  }  </section> |

И представление регистрации:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | @model AuthInternetApplication.Models.RegisterModel  @{      ViewBag.Title = "Регистрация";  }  <hgroup class="title">      <h1>@ViewBag.Title.</h1>      <h2>Создание учетной записи.</h2>  </hgroup>    @using (Html.BeginForm()) {      @Html.AntiForgeryToken()      @Html.ValidationSummary()        <fieldset>          <legend>Форма регистрации</legend>          <ol>              <li>                  @Html.LabelFor(m => m.Email)                  @Html.TextBoxFor(m => m.Email)              </li>              <li>                  @Html.LabelFor(m => m.Password)                  @Html.PasswordFor(m => m.Password)              </li>              <li>                  @Html.LabelFor(m => m.ConfirmPassword)                  @Html.PasswordFor(m => m.ConfirmPassword)              </li>              <li>                  @Html.LabelFor(m => m.FirstName)                  @Html.TextBoxFor(m => m.FirstName)              </li>              <li>                  @Html.LabelFor(m => m.LastName)                  @Html.TextBoxFor(m => m.LastName)              </li>              <li>                  @Html.LabelFor(m => m.Year)                  @Html.TextBoxFor(m => m.Year)              </li>          </ol>          <input type="submit" value="Register" />      </fieldset>  } |

После запуска и регистрации пользователя со всеми данными он окажется в базе данных.

## Использование универсальных провайдеров

Иной подход к авторизации и аутентификации представляют универсальные провайдеры членства и ролей. Хотя в целом они предоставляют все тот же функционал, что и SimpleMembershipProvider и SimpleRolesProvider.

Шаблон Basic для проекта MVC 4 по умолчанию уже включает использование универсальных провайдеров. Если мы создадим новый проект по шаблону Basic, то мы можем увидеть в файле web.config:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | <profile defaultProvider="DefaultProfileProvider">    <providers>      <add name="DefaultProfileProvider" type="System.Web.Providers.DefaultProfileProvider, System.Web.Providers, Version=1.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" connectionStringName="DefaultConnection" applicationName="/" />    </providers>  </profile>  <membership defaultProvider="DefaultMembershipProvider">    <providers>      <add name="DefaultMembershipProvider" type="System.Web.Providers.DefaultMembershipProvider, System.Web.Providers, Version=1.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" connectionStringName="DefaultConnection" enablePasswordRetrieval="false" enablePasswordReset="true" requiresQuestionAndAnswer="false" requiresUniqueEmail="false" maxInvalidPasswordAttempts="5" minRequiredPasswordLength="6" minRequiredNonalphanumericCharacters="0" passwordAttemptWindow="10" applicationName="/" />    </providers>  </membership>  <roleManager defaultProvider="DefaultRoleProvider">    <providers>      <add name="DefaultRoleProvider" type="System.Web.Providers.DefaultRoleProvider, System.Web.Providers, Version=1.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" connectionStringName="DefaultConnection" applicationName="/" />    </providers>  </roleManager>  <sessionState mode="InProc" customProvider="DefaultSessionProvider">    <providers>      <add name="DefaultSessionProvider" type="System.Web.Providers.DefaultSessionStateProvider, System.Web.Providers, Version=1.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" connectionStringName="DefaultConnection" />    </providers>  </sessionState> |

Итак, создадим простейший механизм авторизации и аутентификации с помощью универсальных провайдеров на примере проекта с шаблоном Basic. (Нам не обязательно использовать именно шаблон Basic. Мы можем равным образом использовать и шаблоны Empty и Internet Application, только в этом случае нам необходимо будет добавить в проект пакет *Microsoft.AspNet.Providers*.)

Итак, создадим простейшее приложение по шаблону Basic, назовем его, например, *UniversalProvidersApplication*. По умолчанию он уже содержит определения применяемых провайдеров членства, профилей и ролей в файле web.config, как было показано выше.

Теперь добавим базу данных в папку App\_Data, которая будет использоваться нашим приложением, назвав ее к примеру *StoreDB.mdf*, и изменим строку подключения в файле web.config:

|  |  |
| --- | --- |
| 1  2  3  4 | <connectionStrings>      <add name="DefaultConnection" providerName="System.Data.SqlClient"           connectionString="Data Source=(LocalDB)\v11.0;AttachDbFilename='|DataDirectory|\StoreDB.mdf';Integrated Security=True" />  </connectionStrings> |

Теперь создадим простейшую инфраструктуру для нашего проекта. В начале добавим модели логина и регистрации в папку Models:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46 | using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.ComponentModel.DataAnnotations;    namespace UniversalProvidersApplication.Models  {      public class LogOnModel      {          [Required]          [Display(Name = "Логин")]          public string UserName { get; set; }            [Required]          [DataType(DataType.Password)]          [Display(Name = "Пароль")]          public string Password { get; set; }            [Display(Name = "Запомнить")]          public bool RememberMe { get; set; }      }        public class RegisterModel      {          [Required]          [Display(Name = "Логин")]          public string UserName { get; set; }            [Required]          [DataType(DataType.EmailAddress)]          [Display(Name = "Электронная почта")]          public string Email { get; set; }            [Required]          [StringLength(100, ErrorMessage = "Пароль должен иметь от 6 до 100 символов", MinimumLength = 6)]          [DataType(DataType.Password)]          [Display(Name = "Пароль")]          public string Password { get; set; }            [DataType(DataType.Password)]          [Display(Name = "Подтвердить пароль")]          [Compare("Password", ErrorMessage = "Пароли не совпадают.")]          public string ConfirmPassword { get; set; }      }  } |

Теперь нам нужен контроллер с действиями, который будет обрабатывать ввод пользователя и авторизовывать его, а также представления для ввода данных. Добавим контроллер AccountCotroller со следующим содержимым:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77 | using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.Web.Mvc;  using System.Web.Security;  using UniversalProvidersApplication.Models;    namespace UniversalProvidersApplication.Controllers  {      [AllowAnonymous]      public class AccountController : Controller      {          public ActionResult Login()          {              return View();          }            [HttpPost]          public ActionResult Login(LogOnModel model, string returnUrl)          {              if (ModelState.IsValid)              {                  if (Membership.ValidateUser(model.UserName, model.Password))                  {                      FormsAuthentication.SetAuthCookie(model.UserName, model.RememberMe);                      if (Url.IsLocalUrl(returnUrl))                      {                          return Redirect(returnUrl);                      }                      else                      {                          return RedirectToAction("Index", "Home");                      }                  }                  else                  {                      ModelState.AddModelError("", "Неправильный пароль или логин");                  }              }              return View(model);          }            public ActionResult LogOff()          {              FormsAuthentication.SignOut();              return RedirectToAction("Login", "Account");          }            public ActionResult Register()          {              return View();          }            [HttpPost]          public ActionResult Register(RegisterModel model)          {              if (ModelState.IsValid)              {                  MembershipCreateStatus createStatus;                  Membership.CreateUser(model.UserName, model.Password, model.Email, passwordQuestion: null, passwordAnswer: null, isApproved: true, providerUserKey: null, status: out createStatus);                    if (createStatus == MembershipCreateStatus.Success)                  {                      FormsAuthentication.SetAuthCookie(model.UserName, false);                      return RedirectToAction("Index", "Home");                  }                  else                  {                      ModelState.AddModelError("","Ошибка при регистрации");                  }              }                return View(model);          }      }  } |

Используемый здесь механизм аутентификации форм в принципе описывался ранее и здесь нет ничего нового: при регистрации используется метод Membership.CreateUser для добавления нового пользователя в БД, а затем осуществляем вход, устанавливая куки с помощью FormsAuthentication.SetAuthCookie. При логине с помощью метода Membership.ValidateUser мы проверяем, есть ли пользователь с введенными логином и паролем в системе.

Теперь создадим представления, которые будут регистрировать пользователей и осуществлять вход. Итак, представление регистрации*Register.cshtml*:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51 | @model UniversalProvidersApplication.Models.RegisterModel    @{      ViewBag.Title = "Регистрация";  }    <h2>Регистрация</h2>    @using (Html.BeginForm()) {      @Html.ValidationSummary(true)        <fieldset>          <legend>Введите данные</legend>            <div class="editor-label">              @Html.LabelFor(model => model.UserName)          </div>          <div class="editor-field">              @Html.EditorFor(model => model.UserName)              @Html.ValidationMessageFor(model => model.UserName)          </div>            <div class="editor-label">              @Html.LabelFor(model => model.Email)          </div>          <div class="editor-field">              @Html.EditorFor(model => model.Email)              @Html.ValidationMessageFor(model => model.Email)          </div>            <div class="editor-label">              @Html.LabelFor(model => model.Password)          </div>          <div class="editor-field">              @Html.EditorFor(model => model.Password)              @Html.ValidationMessageFor(model => model.Password)          </div>            <div class="editor-label">              @Html.LabelFor(model => model.ConfirmPassword)          </div>          <div class="editor-field">              @Html.EditorFor(model => model.ConfirmPassword)              @Html.ValidationMessageFor(model => model.ConfirmPassword)          </div>            <p>              <input type="submit" value="Зарегистрировать" />          </p>      </fieldset>  } |

И представление логина *Login.cshtml*:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | @model UniversalProvidersApplication.Models.LogOnModel  @{      ViewBag.Title = "Вход в систему";  }  <h2>Вход в систему</h2>    @using (Html.BeginForm()) {      @Html.ValidationSummary(true)        <fieldset>          <legend>Введите логин и пароль</legend>            <div class="editor-label">              @Html.LabelFor(model => model.UserName)          </div>          <div class="editor-field">              @Html.EditorFor(model => model.UserName)              @Html.ValidationMessageFor(model => model.UserName)          </div>            <div class="editor-label">              @Html.LabelFor(model => model.Password)          </div>          <div class="editor-field">              @Html.EditorFor(model => model.Password)              @Html.ValidationMessageFor(model => model.Password)          </div>            <div class="editor-label">              @Html.LabelFor(model => model.RememberMe)          </div>          <div class="editor-field">              @Html.EditorFor(model => model.RememberMe)              @Html.ValidationMessageFor(model => model.RememberMe)          </div>            <p>              <input type="submit" value="Войти" />          </p>      </fieldset>  } |

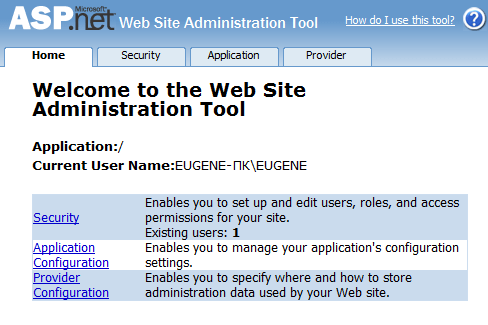
После запуска фреймворк создаст в БД ряд таблиц, предназначенных для хранения учетных записей, ролей и прочей информации, связанной с членством:

Зарегистрируем нового пользователя:

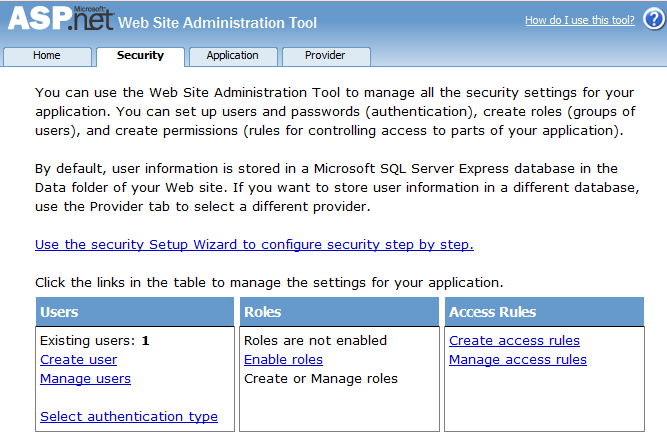
После этого введенная информация попадет в базу данных.

## Web Site Administration Tool

Кроме добавления и изменения настроек учетных записей через формы регистрации, изменения пароля и другие, мы можем использовать средство администрирования **Web Site Administration Tool** - графический интерфейс для управления членство и ролями. Для этого выберем в меню Project пункт ASP.NET Configuration. После этого у нас откроется в веб-браузере страница администрирования веб-приложения:



На вкладке Security мы можем произвести управление учетными записями: создать, изменить или удалить пользователей, добавить роли и установить для пользователей роли:



Для создания нового пользователя выберем ссылку **Create user**. После этого нам откроется форма создания нового пользователя:

Либо мы могли бы выбрать ссылку **Manage users** и изменить информацию об уже добавленном пользователе, применить к нему роли и т.д.:

Чтобы включить роли, нажмем на ссылку *Enable roles*, затем нажмем на появившуюся ссылку *Create or Manage roles* и и нам откроется форма добавления роли:

Поскольку действия в Web Site Administration Tool синхронизированы с бд в проекте, то добавленная роль сразу же появится в таблице Roles (как и добавляемые пользователи в соответствующих таблицах). И после этого мы можем назначать роли пользователю.

Вкладка **Application** предназначена для установки графически настроек приложения, а вкладка **Provider** предназначена для установки провайдеров членства и ролей для приложения.

В завершении разговора о Web Site Administration Tool надо отметить, что этот инструмент не совместим с использованием SimpleMembershipProvider и может использоваться только для при работе с традиционными или универсальными провайдерами членства и ролей.

## Создание своих провайдеров членства и ролей

ASP.NET MVC предоставляет нам законченную модель провайдеров членства/ролей (в виде универсальных провайдеров или в виде SimpleMembershipPrivider API), которые предлагают нам некоторую стандартную инфраструктуру с уже определенными таблицами. Однако иногда стандартные механизмы членства могут не удовлетворять наши потребности. Например, некоторым модель членства кажется довольно тяжеловесной. А кто-то хотел бы применить свою схему таблиц и авторизации. Поэтому в данном случае нам поможет переопределение стандартных механизмов членства и ролей.

Переопределение членства предполагает, что мы должны переопределить методы абстрактного класса **MembershipProvider**, который лежит в основе провайдеров членства. А переопределение провайдера ролей включает реализацию абстрактного класса **RoleProvider**. Посмотрим на пример. Создадим новый проект ASP.NET MVC 4 по шаблону Basic. Назовем его, например, **CustomAuthorization**.

Первым делом определим модели, которые будут описывать учетные записи. Это у нас будут модели пользователей и ролей, а также контекст данных для доступа к ним. Итак, добавим в папку Models следующие классы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.ComponentModel.DataAnnotations;  using System.Data.Entity;    namespace CustomAuthorization.Models  {      public class UserContext : DbContext      {          public DbSet<User> Users { get; set; }          public DbSet<Role> Roles { get; set; }      }        public class User      {          public int Id { get; set; }          public string Email { get; set; }          public string Password { get; set; }          public DateTime CreationDate { get; set; }            public int? RoleId { get; set; }          public Role Role { get; set; }      }        public class Role      {          public int Id { get; set; }          public string Name { get; set; }      }  } |

Во-первых, здесь определен класс пользователя, который будет иметь следующие поля: id, адрес электронной почты, дату регистрации, пароль и ссылку на выполняемую роль в системе. Предполагается, что пользователи в качестве логина на сайте будут использовать адрес электронной почты.

Во-вторых, у нас имеется класс роли в системе, а также контекст UserContext, через который мы будем получать данные из БД.

В отличие от провайдеров членства, рассмотренных в прошлых разделах, здесь мы сами создадим базу данных, структуру которой мы определим сами. Эта структура будет полностью соответствовать вышеописанным моделям.

Итак, создадим новую базу данных для нашего приложения, поместив ее в папку *App\_Data*. Назовем ее, например, **StoreDB**.

Создадим таблицу Roles, которая будет хранить роли пользователей:

Создадим таблицу Users, которая будет хранить пользователей:

Итак, столбцы в таблицах в точности соответствуют определению моделей, а в качестве ссылки на роль в таблице Roles в таблице Users используется внешний ключ. Это и все в плане хранения данных, что нам потребуется для аутентификации и авторизации пользователей.

Добавим в таблицу Roles пару ролей (пусть это будут роли администратора и простого пользователя):

И в файле web.config добавим определение строки подключения:

|  |  |
| --- | --- |
| 1  2  3  4  5 | <connectionStrings>      <add name="UserContext"          connectionString="Data Source=(LocalDB)\v11.0;AttachDbFilename='|DataDirectory|\StoreDB.mdf';Integrated Security=True"          providerName="System.Data.SqlClient" />  </connectionStrings> |

Теперь в следующей теме создадим свой провайдер членства.

## Переопределение провайдера членства

В прошлом разделе мы создали базовую инфраструктуру для собственных провайдеров членства и ролей, а теперь перейдем непосредственно к переопределению провайдера членства. Сначала добавим в проект папку *Providers*, где будут находиться провайдеры членства и ролей. Потом добавим в эту папку новый класс - назовем его **CustomMembershipProvider**. Он будет иметь следующее определение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188  189  190  191  192  193  194  195  196  197  198  199  200  201  202  203  204  205  206  207  208  209  210  211  212  213 | using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.Web.Security;  using System.Web.Helpers;  using System.Security.Cryptography;  using System.Web.WebPages;  using Microsoft.Internal.Web.Utils;  using CustomAuthorization.Models;    namespace CustomAuthorization.Providers  {      public class CustomMembershipProvider : MembershipProvider      {          public override bool ValidateUser(string username, string password)          {              bool isValid = false;                using (UserContext \_db = new UserContext())              {                  try                  {                      User user = (from u in \_db.Users                                   where u.Email == username                                   select u).FirstOrDefault();                        if (user != null &&  Crypto.VerifyHashedPassword(user.Password,password))                      {                          isValid = true;                      }                  }                  catch                  {                      isValid = false;                  }              }              return isValid;          }            public MembershipUser CreateUser(string email, string password)          {              MembershipUser membershipUser = GetUser(email, false);                if (membershipUser == null)              {                  try                  {                      using (UserContext \_db = new UserContext())                      {                          User user = new User();                          user.Email = email;                          user.Password = Crypto.HashPassword(password);                          user.CreationDate = DateTime.Now;                            if(\_db.Roles.Find(2)!=null)                          {                              user.RoleId = 2;                          }                            \_db.Users.Add(user);                          \_db.SaveChanges();                          membershipUser=GetUser(email, false);                          return membershipUser;                      }                  }                  catch                  {                      return null;                  }              }              return null;          }            public override MembershipUser GetUser(string email, bool userIsOnline)          {              try              {                  using (UserContext \_db = new UserContext())                  {                      var users = from u in \_db.Users                                  where u.Email == email                                  select u;                      if (users.Count() > 0)                      {                          User user = users.First();                          MembershipUser memberUser = new MembershipUser("MyMembershipProvider", user.Email, null, null, null, null,                              false, false, user.CreationDate, DateTime.MinValue, DateTime.MinValue, DateTime.MinValue, DateTime.MinValue);                          return memberUser;                      }                  }              }              catch              {                  return null;              }              return null;          }            public override string ApplicationName          {              get              {                  throw new NotImplementedException();              }              set              {                  throw new NotImplementedException();              }          }            public override bool ChangePassword(string username, string oldPassword, string newPassword)          {              throw new NotImplementedException();          }            public override bool ChangePasswordQuestionAndAnswer(string username, string password, string newPasswordQuestion, string newPasswordAnswer)          {              throw new NotImplementedException();          }            public override MembershipUser CreateUser(string username, string password, string email, string passwordQuestion, string passwordAnswer, bool isApproved, object providerUserKey, out MembershipCreateStatus status)          {              throw new NotImplementedException();          }            public override bool DeleteUser(string username, bool deleteAllRelatedData)          {              throw new NotImplementedException();          }            public override bool EnablePasswordReset          {              get { throw new NotImplementedException(); }          }          public override bool EnablePasswordRetrieval          {              get { throw new NotImplementedException(); }          }          public override MembershipUserCollection FindUsersByEmail(string emailToMatch, int pageIndex, int pageSize, out int totalRecords)          {              throw new NotImplementedException();          }          public override MembershipUserCollection FindUsersByName(string usernameToMatch, int pageIndex, int pageSize, out int totalRecords)          {              throw new NotImplementedException();          }          public override MembershipUserCollection GetAllUsers(int pageIndex, int pageSize, out int totalRecords)          {              throw new NotImplementedException();          }          public override int GetNumberOfUsersOnline()          {              throw new NotImplementedException();          }          public override string GetPassword(string username, string answer)          {              throw new NotImplementedException();          }          public override MembershipUser GetUser(object providerUserKey, bool userIsOnline)          {              throw new NotImplementedException();          }          public override string GetUserNameByEmail(string email)          {              throw new NotImplementedException();          }          public override int MaxInvalidPasswordAttempts          {              get { throw new NotImplementedException(); }          }          public override int MinRequiredNonAlphanumericCharacters          {              get { throw new NotImplementedException(); }          }          public override int MinRequiredPasswordLength          {              get { throw new NotImplementedException(); }          }          public override int PasswordAttemptWindow          {              get { throw new NotImplementedException(); }          }          public override MembershipPasswordFormat PasswordFormat          {              get { throw new NotImplementedException(); }          }          public override string PasswordStrengthRegularExpression          {              get { throw new NotImplementedException(); }          }          public override bool RequiresQuestionAndAnswer          {              get { throw new NotImplementedException(); }          }          public override bool RequiresUniqueEmail          {              get { throw new NotImplementedException(); }          }          public override string ResetPassword(string username, string answer)          {              throw new NotImplementedException();          }          public override bool UnlockUser(string userName)          {              throw new NotImplementedException();          }          public override void UpdateUser(MembershipUser user)          {              throw new NotImplementedException();          }      }  } |

Теперь разберем класс провайдера. Во-первых, чтобы реализовать провайдер членства, надо унаследовать его от абстрактного класса MembershipProvider и переопределить его методы. В нашем провайдере в качестве примера мы переопределим всего два метода. Остальные останутся нереализованными.

(Поскольку приложение использует класс шифрования Crypto, то необходимо добавить в проект сборку System.Web.Helpers.dll)

Первый переопределенный метод - **ValidateUser**. Он срабатывает при логине пользователя в системе. В нем мы пытаемся найти пользователя по полученным в качестве параметров логине и паролю с помощью контекста UserContext. Если пользователь найден, то он прошел валидацию.

Второй переопределенный метод - **GetUser**. В нем мы похожим образом пытаемся найти пользователя по введенному логину, только теперь мы возвращаем объект **MebershipUser**, определяющий пользователя членства. При желании можно определить свой класс пользователя членства, но для данного примера сойдет и стандартный. В качестве параметров мы передаем ему свойства пользователя в конструкторе:

|  |  |
| --- | --- |
| 1  2 | MembershipUser memberUser = new MembershipUser("MyMembershipProvider", user.Email, null, null, null, null,                      false, false, user.CreationDate, DateTime.MinValue, DateTime.MinValue, DateTime.MinValue, DateTime.MinValue); |

Тут нас интересуют только пара-тройка параметров - имя провайдера членства, логин (он же email) и дата создания. А параметры со значениями null и DateTime.MinValue в данном случае нам не важны.

Перейдем к третьему определенному в классе методу - **CreateUser**. Он создает при регистрации нового пользователя.

Мы могли бы и переопределить унаследованный метод CreateUser(string username, string password, string email, string passwordQuestion, string passwordAnswer, bool isApproved, object providerUserKey, out MembershipCreateStatus status), передав данные пользователя в виде параметров. Но в данном случае для большей простоты и разнообразия воспользуемся другим способом - создадим свою перегрузку этого метода.

В этом методе новый пользователь через UserContext просто добавляется в БД. Здесь же мы присваиваем роль пользователю. На выходе получаем с помощью ранее переопределенного метода GetUser объект MembershipUser.

Теперь изменим файл web.config, чтобы наше приложение работало с нашим провайдером членства. Во-первых, уберем или закомментируем в этом файле определения других провайдеров:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | <profile defaultProvider="DefaultProfileProvider">    <providers>      <add name="DefaultProfileProvider" type="System.Web.Providers.DefaultProfileProvider, System.Web.Providers, Version=1.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" connectionStringName="DefaultConnection" applicationName="/" />    </providers>  </profile>  <membership defaultProvider="DefaultMembershipProvider">    <providers>      <add name="DefaultMembershipProvider" type="System.Web.Providers.DefaultMembershipProvider, System.Web.Providers, Version=1.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" connectionStringName="DefaultConnection" enablePasswordRetrieval="false" enablePasswordReset="true" requiresQuestionAndAnswer="false" requiresUniqueEmail="false" maxInvalidPasswordAttempts="5" minRequiredPasswordLength="6" minRequiredNonalphanumericCharacters="0" passwordAttemptWindow="10" applicationName="/" />    </providers>  </membership>  <roleManager defaultProvider="DefaultRoleProvider">    <providers>      <add name="DefaultRoleProvider" type="System.Web.Providers.DefaultRoleProvider, System.Web.Providers, Version=1.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" connectionStringName="DefaultConnection" applicationName="/" />    </providers>  </roleManager>  <sessionState mode="InProc" customProvider="DefaultSessionProvider">    <providers>      <add name="DefaultSessionProvider" type="System.Web.Providers.DefaultSessionStateProvider, System.Web.Providers, Version=1.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35" connectionStringName="DefaultConnection" />    </providers>  </sessionState> |

И вместо них добавим определение нашего провайдера:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <membership defaultProvider="MyMembershipProvider">    <providers>      <clear/>      <add name="MyMembershipProvider" type="CustomAuthorization.Providers.CustomMembershipProvider" />    </providers>  </membership> |

И финальная часть - создание контроллера. Добавим в проект в папку Controllers контроллер **AccountCotroller**. Он будет иметь следующий код:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73 | using System;  using System.Web;  using System.Web.Mvc;  using CustomAuthorization.Models;  using System.Web.Security;  using CustomAuthorization.Providers;    namespace CustomAuthorization.Controllers  {      [AllowAnonymous]      public class AccountController : Controller      {          public ActionResult Login()          {              return View();          }          [HttpPost]          public ActionResult Login(LogOnModel model, string returnUrl)          {              if (ModelState.IsValid)              {                  if (Membership.ValidateUser(model.UserName, model.Password))                  {                      FormsAuthentication.SetAuthCookie(model.UserName, model.RememberMe);                      if (Url.IsLocalUrl(returnUrl))                      {                          return Redirect(returnUrl);                      }                      else                      {                          return RedirectToAction("Index", "Home");                      }                  }                  else                  {                      ModelState.AddModelError("", "Неправильный пароль или логин");                  }              }              return View(model);          }          public ActionResult LogOff()          {              FormsAuthentication.SignOut();                return RedirectToAction("Login", "Account");          }            public ActionResult Register()          {              return View();          }            [HttpPost]          public ActionResult Register(RegisterModel model)          {              if (ModelState.IsValid)              {                  MembershipUser membershipUser = ((CustomMembershipProvider)Membership.Provider).CreateUser(model.Email, model.Password);                    if (membershipUser != null)                  {                      FormsAuthentication.SetAuthCookie(model.Email, false);                      return RedirectToAction("Index", "Home");                  }                  else                  {                      ModelState.AddModelError("", "Ошибка при регистрации");                  }              }              return View(model);          }      }  } |

Метод Login имеет стандартную реализацию, только теперь при вызове метода Membership.ValidateUser будет использоваться соответствующий метод нашего провайдера.

В методе Register мы используем код MembershipUser membershipUser = ((CustomMembershipProvider)Membership.Provider).CreateUser(model.Email, model.Password), чтобы получить пользователя членства. Так как здесь использовался не переопределенный метод CreateUser, а его перегрузка, то нам необходимо привести используемый провайдер к типу нашего провайдера: (CustomMembershipProvider)Membership.Provider

Таким образом, после регистрации данные пользователя будут попадать в таблицу Users, определенную в нашей БД, и мы сможем его затем использовать при авторизации и аутентификации.

Если нам при регистрации надо передавать дополнительные параметры в бд, то мы можем просто создать другую перегрузку для метода CreateUser, добавив нужные параметры.

А теперь, используя имеющиеся модели и бд, создадим собственный провайдер ролей.

## Переопределение провайдера ролей

В данном случае мы продолжим работу с проектом, созданным в предыдущем разделе.

Чтобы переопределить провайдер ролей. нам надо создать класс, производный от **RoleProvider**. Поэтому добавим в папку Providers новый класс (назовем его **CustomRoleProvider**) со следующим содержанием:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130 | using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.Web.Security;  using System.Web.Helpers;  using System.Security.Cryptography;  using System.Web.WebPages;  using Microsoft.Internal.Web.Utils;  using CustomAuthorization.Models;    namespace CustomAuthorization.Providers  {      public class CustomRoleProvider : RoleProvider      {          public override string[] GetRolesForUser(string email)          {              string[] role = new string[] { };              using (UserContext \_db = new UserContext())              {                  try                  {                      // Получаем пользователя                      User user = (from u in \_db.Users                                   where u.Email == email                                   select u).FirstOrDefault();                      if (user != null)                      {                          // получаем роль                          Role userRole = \_db.Roles.Find(user.RoleId);                            if (userRole != null)                          {                              role = new string[] { userRole.Name };                          }                      }                  }                  catch                  {                      role = new string[] { };                  }              }              return role;          }          public override void CreateRole(string roleName)          {              Role newRole = new Role() { Name = roleName };              UserContext db = new UserContext();              db.Roles.Add(newRole);              db.SaveChanges();          }          public override bool IsUserInRole(string username, string roleName)          {              bool outputResult = false;              // Находим пользователя              using (UserContext \_db = new UserContext())              {                  try                  {                      // Получаем пользователя                      User user = (from u in \_db.Users                                   where u.Email == username                                   select u).FirstOrDefault();                      if (user != null)                      {                          // получаем роль                          Role userRole = \_db.Roles.Find(user.RoleId);                            //сравниваем                          if (userRole != null && userRole.Name == roleName)                          {                              outputResult = true;                          }                      }                  }                  catch                  {                      outputResult = false;                  }              }              return outputResult;          }          public override void AddUsersToRoles(string[] usernames, string[] roleNames)          {              throw new NotImplementedException();          }            public override string ApplicationName          {              get              {                  throw new NotImplementedException();              }              set              {                  throw new NotImplementedException();              }          }            public override bool DeleteRole(string roleName, bool throwOnPopulatedRole)          {              throw new NotImplementedException();          }            public override string[] FindUsersInRole(string roleName, string usernameToMatch)          {              throw new NotImplementedException();          }            public override string[] GetAllRoles()          {              throw new NotImplementedException();          }            public override string[] GetUsersInRole(string roleName)          {              throw new NotImplementedException();          }            public override void RemoveUsersFromRoles(string[] usernames, string[] roleNames)          {              throw new NotImplementedException();          }            public override bool RoleExists(string roleName)          {              throw new NotImplementedException();          }      }  } |

Здесь в целях демонстрации переопределено три метода. Первый из них - GetRolesForUser позволяет получать набор ролей для определенного пользователя.

Второй метод - CreateRole - предполагает создание роли.

И третий переопределенный метод - IsUserInRole - указывает, выполняет ли пользователь определенную роль в системе.

В принципе здесь опять, как и в случае с провайдером членства, идет стандартная работа с моделями Role и User через контекст данных UserContext, и ничего очень сложного не должно быть.

Теперь используем провайдер ролей в приложении. Сначала добавим в файл *web.config* определение провайдера ролей рядом с определением провайдера членства (либо в другом месте в пределах узла system.web:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <roleManager enabled="true" defaultProvider="MyRoleProvider">    <providers>      <clear/>      <add name="MyRoleProvider" type="CustomAuthorization.Providers.CustomRoleProvider" />    </providers>  </roleManager> |

Теперь мы можем использовать наш провайдер ролей как и любой другой. Например, определим в каком-нибудь контроллере разный доступ к его ресурсам в зависимости от роли:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.Web.Mvc;    namespace CustomAuthorization.Controllers  {      [Authorize]      public class HomeController : Controller      {          public string Index()          {              return "Ваш логин: " + HttpContext.User.Identity.Name;          }          [Authorize(Roles="admin")]          public string ViewRole()          {              return "Ваша роль: Администратор";          }      }  } |

Для тестирования можно зарегистрировать пару пользователь с разными ролями и попробовать получить доступ к действиям контроллера.

## Провайдер профилей и его переопределение

Если провайдер членства помогает системе определить, что это за пользователь, его идентифицировать, провайдер ролей указывает системе на статус пользователя и наделяет его определенные правами доступа, то провайдер профилей определят набор дополнительных свойств пользователя. К подобным дополнительным свойствам могут относиться, например, рост, возраст, вес, должность и т.д.

Использовать профили очень просто:

|  |  |
| --- | --- |
| 1  2  3  4  5 | // получение  ViewBag.Name = HttpContext.Profile["Name"];  // установка  string name = "Bill Gates";  HttpContext.Profile["Name"]=name; |

Однако если вы так напишите в приложении, то скорее всего получите ошибку, так как перед использованием свойство Name должно быть определено. Теперь посмотрим, как все это должно быть определено и как все это использовать.

При использовании универсальных провайдеров (например, в проектах по шаблону Basic) у нас в файле *web.config* уже подключен стандартный провайдер профилей:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <profile defaultProvider="DefaultProfileProvider">      <providers>        <add name="DefaultProfileProvider" type="System.Web.Providers.DefaultProfileProvider, System.Web.Providers,              Version=1.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35"              connectionStringName="DefaultConnection" applicationName="/" />      </providers>  </profile> |

Добавим в узел *profile* определение профиля, изменив данный узел следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | <profile defaultProvider="DefaultProfileProvider">        <providers>          <add name="DefaultProfileProvider" type="System.Web.Providers.DefaultProfileProvider,          System.Web.Providers, Version=1.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35"          connectionStringName="DefaultConnection" applicationName="/" />        </providers>        <properties>          <add name="FirstName" type="System.String" />          <add name="LastName" type="System.String" />          <add name="Age" type="System.Int32"/>        <group name="Contact">          <add name="Website" type="System.String"/>          <add name="Email" type="System.String"/>        </group>        </properties>  </profile> |

Определение свойств профиля помещается в узел properties. Определение свойства заключается в использовании элемента add, в котором мы определяем название свойства через атрибут name и его тип через атрибут type. В качестве типа могут выступать примитивные типы .NET.

Здесь также мы можем определить подгруппу свойств с помощью элемента group. В дальнейшем мы можем использовать данные свойства профиля:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | [Authorize]  public class HomeController : Controller  {      public string Index()      {          HttpContext.Profile["FirstName"] = "Иван";          HttpContext.Profile["LastName"] = "Петров";            return "Добро пожаловать, " + HttpContext.Profile["FirstName"] + "  " +              HttpContext.Profile["LastName"];      }  } |

Определив в файле web.config свойства профиля, мы можем к ним обратиться, например присвоить: HttpContext.Profile["FirstName"] . Причем до присвоения эти свойства ничего не несут. А затем мы обратно можем получить.

Возникает вопрос, а куда это все сохраняется? При работе с универсальными провайдерами у нас создается в базе данных автоматически набор таблиц, одна из которых предназначена для хранения профилей - **Profiles**.

Перед использованием профилей она пуста и ничего в ней нет. Но после присвоения в ней появляются соответствующие данные. Например, в моем случае это будут следующие данные:

Названия установленных свойств у нас оказываются в поле PropertyNames, а их значения - в поле PropertyValueStrings. Кроме того, в таблице еще есть ряд столбцов, из которых хотелось бы выделить UserId. Это поле содержит ключ пользователя, которое имеет по этому полю связано с другими таблицами. А в качестве пользователя выступает текущий залогинившийся пользователь, который вошел в систему и обратился к методу Index контроллера Home.

Чтобы обратиться к свойствам, которые указаны в группе, мы обращаемся к ним через эту группу:

|  |  |
| --- | --- |
| 1  2 | HttpContext.Profile.GetProfileGroup("Contact")["Email"] = "ivpetrov5@mail.ru";  ViewBag.Email = HttpContext.Profile.GetProfileGroup("Contact")["Email"]; |

### Переопределение провайдера профилей

В данном случае мы продолжим работу с проектом, где мы переопределяли универсальные провайдеры членства и ролей. Для начала добавим модель, которая будет содержать все свойства профиля. Пусть эта модель будет называться **Profile**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class Profile  {      public int Id { get; set; }        // Внешний ключ для связи с пользователем      public int UserId { get; set; }      public virtual User User { get; set; }        public string FirstName { get; set; }      public string LastName { get; set; }      public int Age { get; set; }      public DateTime LastUpdateDate { get; set; }  } |

И изменим класс контекста данных, чтобы он мог получать данные профилей:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class UserContext : DbContext  {      public DbSet<User> Users { get; set; }      public DbSet<Role> Roles { get; set; }      public DbSet<Profile> Profiles { get; set; }  } |

Теперь нам нужно место, где мы будем хранить данные профилей. Для этого в базе данных создадим таблицу Profiles со следующим определением:

CREATE TABLE [dbo].[Profiles]

(

[Id] INT NOT NULL PRIMARY KEY IDENTITY,

[UserId] INT NOT NULL,

[FirstName] NVARCHAR(50) NULL,

[LastName] NVARCHAR(50) NULL,

[Age] INT NULL,

[LastUpdateDate] DATETIME NULL,

CONSTRAINT [FK\_Profiles\_ToUsers] FOREIGN KEY ([UserId]) REFERENCES [Users]([Id]) ON DELETE CASCADE

)

Таким образом, основной таблице у нас является Users, где находятся данные id, логина и пароля пользователей. Чтобы связать профиль с определенным пользователем в таблице профилей, мы определяем поле UserId. И это поле будет хранить значение столбца id из таблицы Users.

А теперь к самому интересному. Добавим в папку Providers класс провайдера профилей (назовем его **CustomProfileProvider**), который будет иметь следующее определение:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144 | using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.Web.Profile;  using System.Configuration;  using CustomAuthorization.Models;  using System.Data;    namespace CustomAuthorization.Providers  {      public class CustomProfileProvider : ProfileProvider      {          public override SettingsPropertyValueCollection GetPropertyValues(SettingsContext context, SettingsPropertyCollection collection)          {              // коллекция, которая возвращает значения свойств профиля              SettingsPropertyValueCollection result = new SettingsPropertyValueCollection();                if (collection == null || collection.Count < 1 || context == null)              {                return result;              }              // получаем из контекста имя пользователя - логин в системе              string username = (string)context["UserName"];              if (String.IsNullOrEmpty(username))                  return result;                UserContext db = new UserContext();              // получаем id пользователя из таблицы Users по логину              int userId = db.Users.Where(u => u.Email.Equals(username)).FirstOrDefault().Id;              // по этому id извлекаем профиль из таблицы профилей              Profile profile = db.Profiles.Where(u => u.UserId==userId).FirstOrDefault();              if (profile != null)              {                  foreach (SettingsProperty prop in collection)                  {                      SettingsPropertyValue svp = new SettingsPropertyValue(prop);                      svp.PropertyValue = profile.GetType().GetProperty(prop.Name).GetValue(profile, null);                      result.Add(svp);                  }              }              else              {                  foreach (SettingsProperty prop in collection)                  {                      SettingsPropertyValue svp = new SettingsPropertyValue(prop);                      svp.PropertyValue = null;                      result.Add(svp);                  }              }              return result;          }            public override void SetPropertyValues(SettingsContext context, SettingsPropertyValueCollection collection)          {              // получаем логин пользователя              string username = (string)context["UserName"];                if (username == null || username.Length < 1 || collection.Count < 1)                  return;                UserContext db = new UserContext();              // получаем id пользователя из таблицы Users по логину              int userId = db.Users.Where(u => u.Email.Equals(username)).FirstOrDefault().Id;              // по этому id извлекаем профиль из таблицы профилей              Profile profile = db.Profiles.Where(u => u.UserId == userId).FirstOrDefault();              // если такой профиль уже есть изменяем его              if (profile != null)              {                  foreach (SettingsPropertyValue val in collection)                  {                      profile.GetType().GetProperty(val.Property.Name).SetValue(profile, val.PropertyValue);                  }                  profile.LastUpdateDate = DateTime.Now;                  db.Entry(profile).State = EntityState.Modified;              }              else              {                  // если нет, то создаем новый профиль и добавляем его                  profile = new Profile();                  foreach (SettingsPropertyValue val in collection)                  {                      profile.GetType().GetProperty(val.Property.Name).SetValue(profile, val.PropertyValue);                  }                  profile.LastUpdateDate = DateTime.Now;                  profile.UserId = userId;                  db.Profiles.Add(profile);              }              db.SaveChanges();          }            public override int DeleteInactiveProfiles(ProfileAuthenticationOption authenticationOption, DateTime userInactiveSinceDate)          {              throw new NotImplementedException();          }            public override int DeleteProfiles(string[] usernames)          {              throw new NotImplementedException();          }            public override int DeleteProfiles(ProfileInfoCollection profiles)          {              throw new NotImplementedException();          }            public override ProfileInfoCollection FindInactiveProfilesByUserName(ProfileAuthenticationOption authenticationOption, string usernameToMatch, DateTime userInactiveSinceDate, int pageIndex, int pageSize, out int totalRecords)          {              throw new NotImplementedException();          }            public override ProfileInfoCollection FindProfilesByUserName(ProfileAuthenticationOption authenticationOption, string usernameToMatch, int pageIndex, int pageSize, out int totalRecords)          {              throw new NotImplementedException();          }            public override ProfileInfoCollection GetAllInactiveProfiles(ProfileAuthenticationOption authenticationOption, DateTime userInactiveSinceDate, int pageIndex, int pageSize, out int totalRecords)          {              throw new NotImplementedException();          }            public override ProfileInfoCollection GetAllProfiles(ProfileAuthenticationOption authenticationOption, int pageIndex, int pageSize, out int totalRecords)          {              throw new NotImplementedException();          }            public override int GetNumberOfInactiveProfiles(ProfileAuthenticationOption authenticationOption, DateTime userInactiveSinceDate)          {              throw new NotImplementedException();          }            public override string ApplicationName          {              get              {                  throw new NotImplementedException();              }              set              {                  throw new NotImplementedException();              }          }      }  } |

Класс провайдера наследуется от класса ProfileProvider и содержит определения ряда его методов. Так как у нас будет самый примитивный провайдер профилей, то мы реализуем всего два его метода: **SetPropertyValues** и **GetPropertyValues**.

Метод GetPropertyValues принимает два параметра - контекст (откуда мы можем получить разные данные типа логина пользователя) и коллекцию свойств профиля, которые будут использоваться. На выходе мы возвращаем объект SettingsPropertyValueCollection, который содержит названия свойств и их значения. Важно в любом случае передать названия свойств, даже если свойства не несут никакого значения, иначе среда не сможет определить свойство и присвоить ему значение (например, HttpContext.Profile["FirstName"] = "Vova";).

В самом методе мы получаем пользователя, его id, профиль и из профиля передаем значение в выходную коллекцию.

Метод SetPropertyValues берет установленные данные и уже через контекст передает их в базу данных.

Теперь подключим провайдер в файле *web.config*, добавив после определения провайдера ролей следующие строки:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | <profile enabled="true" defaultProvider="MyProfileProvider">    <providers>      <add  name="MyProfileProvider" type="CustomAuthorization.Providers.CustomProfileProvider"/>    </providers>    <properties>      <add name="FirstName" type="System.String" />      <add name="LastName" type="System.String" />      <add name="Age" type="System.Int32"/>    </properties>  </profile> |

И в завершении мы можем его протестировать, например, прописав в контроллере Home следующий метод:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | public string ViewProfile()  {      HttpContext.Profile["FirstName"] = "Vova";      HttpContext.Profile["LastName"] = "Pupkin";      HttpContext.Profile.SetPropertyValue("Age", 28);        return "Имя Фамилия: " + HttpContext.Profile["FirstName"].ToString() +          " " + HttpContext.Profile["LastName"].ToString() + " возраст: "+          HttpContext.Profile["Age"].ToString();  } |

Установка свойств может происходить как напрямую (HttpContext.Profile["FirstName"] = "Vova"), так и с помощью метода SetPropertyValue. После установки данные профиля появятся в таблице Profiles в БД, и затем мы сможем их оттуда получать и использовать в приложении.

## OAuth-аутентификация

В современном мире велика популярность различных социальных сервисов и соцсетей - google+, twitter, facebook и т.д. Это привело разработчиков к мысли использовать в качестве логина на сайт учетную запись в данных сервисах и соцсетях. До MVC4 это сложно было сделать. Но в MVC4 появились соответствующие механизмы, которые позволяют предельно упростить аутентификацию с помощью соцсетей.

В ряде случаев нам надо будет зарегистрировать свое приложение в социальных сервисах, чтобы воспользоваться возможностью подобной авторизации (как например в Facebook). А некоторые сервисы как Goggle и Yohoo, позволяют использовать возможность аутентификации на основе социальных сервисов без всякой регистрации.

Итак, в новом проекте по шаблону Internet Apllication мы уже можем найти уже готовую инфраструктуру для использования OAuth-аутентификации. Если вы посмотрите на файл **AuthConfig.cs**, который находится в папке App\_Start, то вы найдете в нем такой код:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using Microsoft.Web.WebPages.OAuth;  using SocialAuthorization.Models;    namespace SocialAuthorization  {      public static class AuthConfig      {          public static void RegisterAuth()          {              // To let users of this site log in using their accounts from other sites such as Microsoft, Facebook, and Twitter,              // you must update this site. For more information visit <http://go.microsoft.com/fwlink/?LinkID=252166>                //OAuthWebSecurity.RegisterMicrosoftClient(              //    clientId: "",              //    clientSecret: "");                //OAuthWebSecurity.RegisterTwitterClient(              //    consumerKey: "",              //    consumerSecret: "");                //OAuthWebSecurity.RegisterFacebookClient(              //    appId: "",              //    appSecret: "");                //OAuthWebSecurity.RegisterGoogleClient();          }      }  } |

Так, настройка аутентификации через Facebook:

|  |  |
| --- | --- |
| 1  2  3 | //OAuthWebSecurity.RegisterFacebookClient(  //    appId: "",  //    appSecret: ""); |

Если ваше веб-приложение зарегистрировано в соцсети facebook, то вы затем можете ввести в параметр appId идентификатор вашего приложения в этой сети, а в параметр appSecret секретное слово. Значения для обоих параметров выдаются уже самой соцсетью при регистрации приложения.

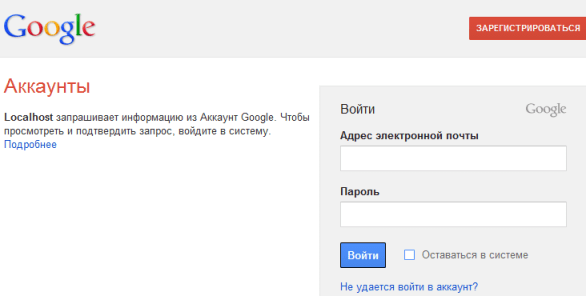
В данном случае воспользуемся аутентификации с помощью Google и закомментируем строку OAuthWebSecurity.RegisterGoogleClient();.

Контроллер AccoutController уже содержит ряд действий для работы с подобной схемой авторизации: ExternalLogin (для авторизации с помощью внешних сервисов), Manage (для управления учетной записью), ExternalLoginConfirmation (для подтверждения авторизации) и ряд других.

Затем в папке *Views/Account* можно увидеть ряд представлений, предназначенных для работы с OAuth. Например, частичное представление*\_ExternalLoginListPartial.cshtml* - здесь можно настроить отображение списка сервисов, *ExternalLoginConfirmation* - для отображения формы подтверждения логина и *ExternalLoginFailure* - для отображения формы ошибки авторизации.

При обращении к странице логина после этого у нас будет выбор способа авторизации - с помощью стандартных механизмов членства, либо с помощью учетной записи Google.

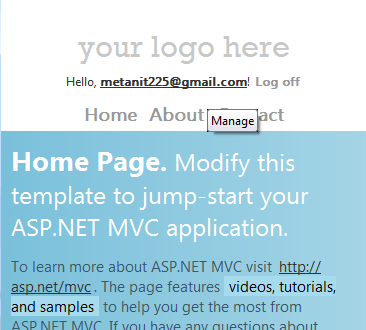
Нажмем на кнопку Google и мы попадем на страницу авторизации в самом сервисе google.



Введем логин и пароль, и после успешной авторизации на google нас попросят принять некоторые разрешения:

Нажмем на кнопку Принять и после этого мы попадем на страницу нашего приложения, где нам надо будет ассоциировать свою учетную запись внутри нашего приложения с учетной записью на google.

Нажмем на кнопку Register и тем самым свяжем учетные записи.



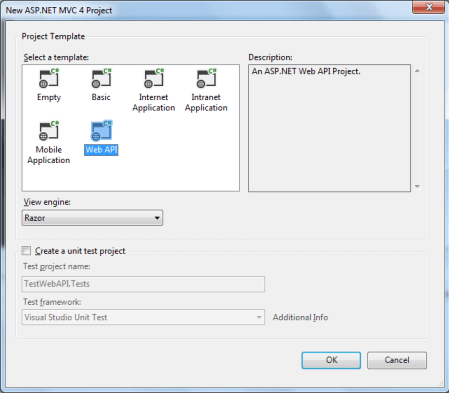
# Web API

## Введение в Web API

Одним из преимуществ стека технологий на платформе .NET является возможность создания сервисов. Так, в ASP.NET мы могли создавать ASMX-веб-службы. Кроме того, мы можем использовать более функциональную в этом плане технологию WCF для создания веб-служб.

Но с последним релизом MVC 4 и вообще всей платформы .NET 4.5 в нашем распоряжении оказался еще один инструмент для создания веб-служб - Web API. Концепция Web API (которую кстати можно использовать не только в MVC, но и в веб-формах) новый подход к реализации веб-приложений.

Создадим простое приложение ASP.NET MVC 4, указав в качестве шаблона проекта **Web API**:



Среда создаст обычный mvc-проект со стандартной структурой. По умолчанию нам будет создано два контроллера. Один из них стандартный контроллер HomeController. Второй контроллер - **ValuesController**, который и реализует функционал Web API:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | public class ValuesController : ApiController  {      // GET api/values      public IEnumerable<string> Get()      {          return new string[] { "value1", "value2" };      }        // GET api/values/5      public string Get(int id)      {          return "value";      }        // POST api/values      public void Post([FromBody]string value)      {      }        // PUT api/values/5      public void Put(int id, [FromBody]string value)      {      }        // DELETE api/values/5      public void Delete(int id)      {      }  } |

Как видно, определение контроллера Web API отличается от обычного контроллера. Во-первых, он образован от класса ApiController, который не связан с базовым классом обычных контроллеров - Controller

Во-вторых, контроллеры Web API применяют стиль REST. Вообще поддержка архитектуры REST (Representation State Transfer или "передача состояния представления") является одним из основных узловых пунктов технологии Web API.

Для взаимодействия с сервером в REST-архитектуре используются методы HTTP:

* GET
* POST
* PUT
* DELETE

У нас тут нет обычных методов действий, как в традиционных контроллерах, которые возвращают ActionResult. А определенные в контроллере ValuesContoller методы сопоставляются с одноименными методами HTTP.

### Маршрутизация в Web API

Поскольку в Web API методы контроллера не являются прямыми ресурсами и сопоставляются с методами HTTP, то и весь механизм маршрутизации действует не как при определении обычных маршрутов. Все определения маршрутов для Web API находятся в файле*WebApiConfig.cs* (в папке *App\_Start*):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public static class WebApiConfig  {      public static void Register(HttpConfiguration config)      {          config.Routes.MapHttpRoute(              name: "DefaultApi",              routeTemplate: "api/{controller}/{id}",              defaults: new { id = RouteParameter.Optional }          );      }  } |

В данном случае определен один маршрут, где в качестве второго параметра выступает контроллер, а третий необязательный параметр представляет некоторый идентификатор. Таким образом, в отличие от маршрутов обычных контроллеров у нас здесь нет действия, только контроллер и дополнительный необязательный параметр

В итоге обращение *api/values* будет соответствовать обращению к контроллеру ValuesCotroller, причем почти ко всем действиям сразу (кроме Get(int id) - так как в данном случае необходим еще идентификатор, например *api/values/2*)

Но, как уже выше говорилось, в зависимости от использованного метода HTTP фреймворк будет различать к какому именно действию относится текущий запрос.

Так, например, запрос GET *api/values* будет сопоставлен с методом IEnumerable<string> Get() и вернет в ответ браузеру коллекцию элементов string.

Если же сервер получит запрос PUT с адресом *api/values*, то такой запрос будет сопоставлен с методом Put.

А запросу GET *api/values/7* будет соответствовать метод string Get(int id), так как этот метод принимает параметр.

### Условности при наименовании методов

При создании методов контроллера Web API действует некоторые условности. Так, имена методов по умолчанию должны начинаться с имени предназначенного для них метода HTTP. В случае с контроллером по умолчанию все просто: все методы действий носят названия методов HTTP.

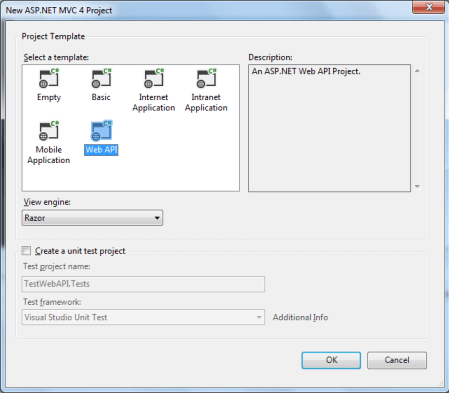
Однако мы можем использовать и любые другие имена без префиксов, но в этом случае нам надо будет явно указать метод HTTP в виде атрибута, например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | public class ValuesController : ApiController  {      public IEnumerable<string> GetAllItems()      {          return new string[] { "value1", "value2" };      }        public string GetItem(int id)      {          return "value";      }        [HttpPost]      public void CreateItem([FromBody]string value)      {      }        [HttpPut]      public void EditItem(int id, [FromBody]string value)      {      }        [HttpDelete]      public void RemoveItem(int id)      {      }  } |

Таким образом, если первые два действия соответствуют соглашениям о наименовании (в начале содержат название метода HTTP), то к остальным мы применяем атрибуты, чтобы система знала, с каким методом надо сопоставлять запрос.

## Создание и тестирование приложения Web API

Итак, создадим новое приложение MVC 4 по типу Web API:



Назовем его, его например, BookStore. Допустим, наше приложение будет работать с книгами.

Сначала создадим модель данных, описывающую книгу. Добавим в папку Models новый класс Book:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class Book  {      public int Id { get; set; }      public string Name { get; set; }      public string Author { get; set; }      public int Year { get; set; }  } |

Затем добавим в папку Models контекст данных BookContext, через который будем взаимодействовать с базой данных:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | using System.Data.Entity;    namespace BookStore.Models  {      public class BookContext : DbContext      {          public DbSet<Book> Books { get; set; }      }  } |

И добавим в папку *App\_Data* базу данных для нашего приложения, где будут храниться данные. В моем случае база данных будет называться BookDB. Создадим в ной базе данных таблицу Books для моделей книг:

И наполним таблицу некоторыми данными:

И в завершении настройки модельной части изменим строку подключения в файле *web.config*:

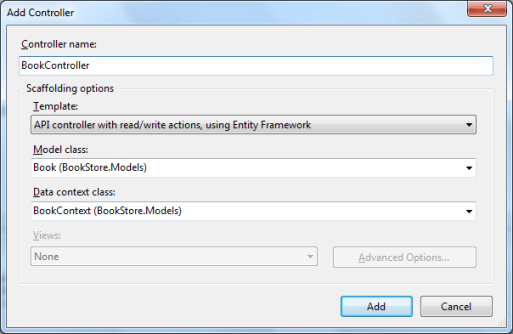
|  |  |
| --- | --- |
| 1  2  3  4 | <connectionStrings>      <add name="BookContext" providerName="System.Data.SqlClient"          connectionString="Data Source=(LocalDB)\v11.0;AttachDbFilename='|DataDirectory|\BookDB.mdf';Integrated Security=True" />  </connectionStrings> |

Теперь перейдем к созданному по умолчанию контроллеру ValuesController и изменим его следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56 | using System;  using System.Collections.Generic;  using System.Linq;  using System.Data.Entity;  using System.Net;  using System.Net.Http;  using System.Web.Http;  using BookStore.Models;  using System.Data;    namespace BookStore.Controllers  {      public class ValuesController : ApiController      {          BookContext db = new BookContext();            public IEnumerable<Book> GetBooks()          {              return db.Books;          }            public Book GetBook(int id)          {              Book book = db.Books.Find(id);              return book;          }            [HttpPost]          public void CreateBook([FromBody]Book book)          {              db.Books.Add(book);              db.SaveChanges();          }            [HttpPut]          public void EditBook(int id, [FromBody]Book book)          {              if (id == book.Id)              {                  db.Entry(book).State = EntityState.Modified;                    db.SaveChanges();              }          }            public void DeleteBook(int id)          {              Book book = db.Books.Find(id);              if(book!=null)              {                  db.Books.Remove(book);                  db.SaveChanges();              }          }      }  } |

Таким образом, у нас получился практически стандартный контроллер, реализующий CRUD-модель, то есть добавление, получение, обновление и удаление данных.

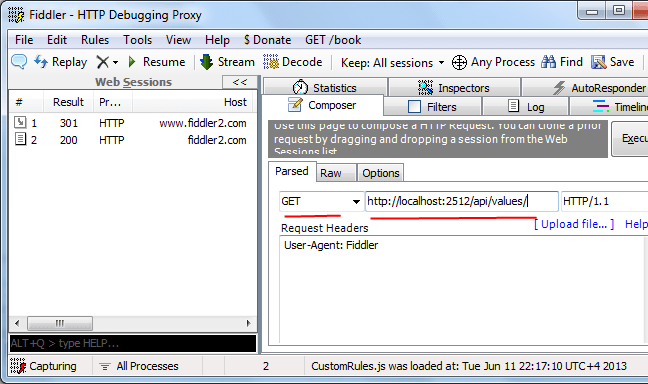
Кстати чтобы создать аналогичный функционал мы можем не писать вручную код, а воспользоваться шаблонами формирования при создании контроллера. Для этого просто при создании нового API-контроллера укажем модель и класс контекста данных, которые будут использоваться (перед использованием модели приложение необходимо скомпилировать):



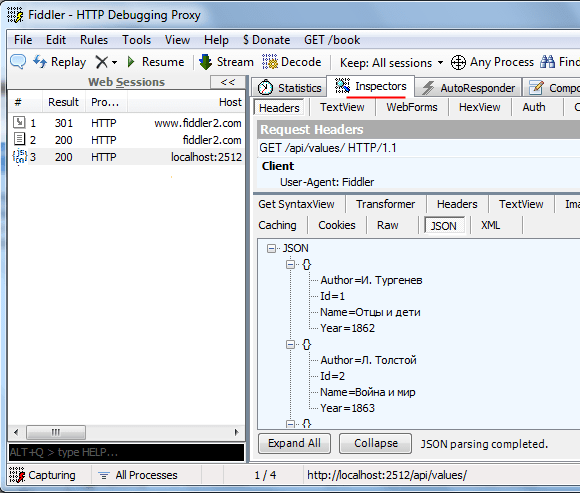
И шаблон формирование создаст нам аналогичный функционал с небольшими изменениями в коде.

Итак, теперь мы можем перейти к построению клиентской части, но перед этим протестируем приложение. Для тестирования GET-запросов нам достаточно простого веб-браузера, но воспользуемся более удобным инструментом web-отладки как **Fiddler**.

[Fiddler](http://fiddler2.com/) позволяет отправлять различные запросы к веб-ресурсам. Итак, запустим наш проект в Visual Studio на выполнение. Параллельно откроем программу Fiddler. После запуска проекта должен запуститься встроенный asp.net сервер, который хостирует наше приложение. В Fiddler перейдите на вкладку Composer и введите в поле адреса строку запроса к нашему сервису. Например, в моем случае локальный сервер запустился по адресу *http://localhost:2512/*, поэтому строка запроса будет выглядеть так: *http://localhost:2512/api/values/*. Поскольку у нас слева от адреса стоит метод GET, поэтому в данном случае мы будем осуществлять запрос к методу public IEnumerable<Book> GetBooks().



После ввода адреса нажмем на Enter и перейдем на вкладку Inspectors, выбрав JSON в качестве формата отображения ответа, то увидим результат запроса - а именно список книг, который Fiddler получает в сериализованном виде:



Таким образом, мы убедились, что приложение работает. Подобным образом мы можем задать в Fiddler и другие запросы. А теперь перейдем к построению клиентской части приложения.

## Создание представления для Web API

Для создания представления возьмем уже имеющее в нашем проекте представление *Index.cshtml*.

Наш визуальный интерфейс будет очень прост: на странице будет три блока. Первый блок (tableBlock) будет предназначен для вывода информации обо всех книгах, второй блок (editBlock) будет содержать форму редактирования книги, а третий блок (createBlock) будет включать форму добавления новой книги. При этом второй и третий блоки будут перекрывать друг друга и в зависимости от того, что нам надо - создать новую или редактировать имеющуюся книгу, мы будем делать видимым один из блоков, управляя css-стилем display.

Итак, изменим файл *Index.cshtml* следующим образом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169 | <div id="tableBlock"></div>  <div id="editBlock"><h3>Редактирование книги</h3>      <table>      <tr><td><input type="hidden" id="editId" /></td><td></td></tr>      <tr><td><label>Название: </label></td><td><input type="text" id="editName" /></td></tr>      <tr><td><label>Автор: </label></td><td><input type="text" id="editAuthor" /></td></tr>      <tr><td><label>Год: </label></td><td><input type="number" id="editYear" /></td></tr>          </table>      <button id="editBook">Сохранить</button>  </div>  <div id="createBlock">      <h3>Добавление книги</h3>      <table>      <tr><td><label>Название: </label></td><td><input type="text" id="addName" /></td></tr>      <tr><td><label>Автор: </label></td><td><input type="text" id="addAuthor" /></td></tr>      <tr><td><label>Год: </label></td><td><input type="number" id="addYear" /></td></tr>       </table>      <button id="addBook">Сохранить</button>  </div>  <script type="text/javascript">      $(document).ready(function () {            GetAllBooks();            $("#editBook").click(function (event) {              event.preventDefault();              EditBook();          });            $("#addBook").click(function (event) {              event.preventDefault();              AddBook();          });        });      // Получение всех книг по ajax-запросу      function GetAllBooks() {            $("#createBlock").css('display', 'block');          $("#editBlock").css('display', 'none');          $.ajax({              url: '<http://localhost:2512/api/values>',              type: 'GET',              dataType: 'json',              success: function (data) {                  WriteResponse(data);              },              error: function (x, y, z) {                  alert(x + '\n' + y + '\n' + z);              }          });      }   // Добавление новой книги      function AddBook() {          // получаем значения для добавляемой книги          var book = {              Name: $('#addName').val(),              Author: $('#addAuthor').val(),              Year: $('#addYear').val()          };            $.ajax({              url: '/api/values',              type: 'POST',              data: JSON.stringify(book),              contentType: "application/json;charset=utf-8",              success: function (data) {                  GetAllBooks();              },              error: function (x, y, z) {                  alert(x + '\n' + y + '\n' + z);              }          });      }   // Удаление книги      function DeleteBook(id) {            $.ajax({              url: '/api/values/' + id,              type: 'DELETE',              contentType: "application/json;charset=utf-8",              success: function (data) {                  GetAllBooks();              },              error: function (x, y, z) {                  alert(x + '\n' + y + '\n' + z);              }          });      }      // редактирование книги      function EditBook() {          var id = $('#editId').val()          // получаем новые значения для редактируемой книги          var book = {              Id: $('#editId').val(),              Name: $('#editName').val(),              Author: $('#editAuthor').val(),              Year: $('#editYear').val()          };          $.ajax({              url: '/api/values/' + id,              type: 'PUT',              data: JSON.stringify(book),              contentType: "application/json;charset=utf-8",              success: function (data) {                  GetAllBooks();              },              error: function (x, y, z) {                  alert(x + '\n' + y + '\n' + z);              }          });      }   // вывод полученных данных на экран      function WriteResponse(books) {          var strResult = "<table><th>ID</th><th>Название</th><th>Автор</th><th>Год издания</th>";          $.each(books, function (index, book) {              strResult += "<tr><td>" + book.Id + "</td><td> " + book.Name + "</td><td>" +              book.Author + "</td><td>" + book.Year +              "</td><td><a id='editItem' data-item='" + book.Id + "' onclick='EditItem(this);' >Редактировать</a></td>" +              "<td><a id='delItem' data-item='" + book.Id + "' onclick='DeleteItem(this);' >Удалить</a></td></tr>";          });          strResult += "</table>";          $("#tableBlock").html(strResult);        }      // обработчик удаления      function DeleteItem(el) {            // получаем id удаляемого объекта          var id = $(el).attr('data-item');          DeleteBook(id);      }      // обработчик редактирования      function EditItem (el) {            // получаем id редактируемого объекта          var id = $(el).attr('data-item');          GetBook(id);      }      // вывод данных редактируемой книги в поля для редактирования      function ShowBook(book) {          if (book != null) {              $("#createBlock").css('display', 'none');              $("#editBlock").css('display', 'block');              $("#editId").val(book.Id);              $("#editName").val(book.Name);              $("#editAuthor").val(book.Author);              $("#editYear").val(book.Year);          }          else {              alert("Такая книга не существует");          }      }      // запрос книги на редактирование      function GetBook(id) {            $.ajax({              url: '/api/values/' + id,              type: 'GET',              dataType: 'json',              success: function (data) {                  ShowBook(data);              },              error: function (x, y, z) {                  alert(x + '\n' + y + '\n' + z);              }          });      }  </script> |

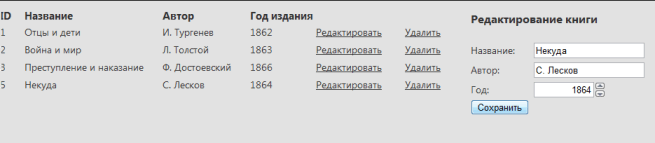
Так как в моем случае приложение развернуто на локальном сервере по адресу http://localhost:xxxx, то а качестве адреса для отправки ajax-запросов используется относительный url */api/values/*

Поскольку в данном случае мы используем код jquery, то необходимо убедиться, что на мастер странице уже подключена библиотека jquery, по умолчанию она подключается внизу страницы. Также добавим в файл стилей определения стилей для блоков:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | div {      display:inline-block;      margin:20px;  }  #tableBlock {      width:auto;  }  #editBlock {      display:none;      float:right;  }  #createBlock {      float:right;  } |

В итоге после загрузке страницы в веб-браузере мы получим все наши книги из БД:

Мы можем отредактировать уже имеющуюся книгу, нажав на ссылку 'Редактировать' у соответствующей книги. И тогда у нас в станет видимой форма для редактирования, поля которой будут заполнены значениями этой книги:



Теперь разберем код представления. При загрузке страницы в браузере в коде javascript мы одновременно получаем все книги из БД с помощью функции GetAllBooks:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | function GetAllBooks() {      $("#createBlock").css('display', 'block');      $("#editBlock").css('display', 'none');      $.ajax({          url: '/api/values',          type: 'GET',          dataType: 'json',          success: function (data) {              WriteResponse(data);          },          error: function (x, y, z) {              alert(x + '\n' + y + '\n' + z);          }      });  } |

Функция переключает видимость блоков createBlock и editBlock и отправляет запрос на сервер. Поскольку это запрос GET без параметра, он будет сопоставлен с методом контроллера GetBooks, а полученные из этого метода данные будут выведены на страницу через функцию WriteResponse.

В функции WriteResponse на каждую ссылку 'Редактировать' мы вешаем обработчик нажатия EditItem(this), а на ссылку 'Удалить' - обработчик DeleteItem(this). Оба эти обработчика в качестве параметра получают нажатый элемент <a...> и затем получают через атрибут data-item id нажатой книги.

Обработчик DeleteItem вызывает метод DeleteBook, передавая в него id удаляемой книги:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | function DeleteBook(id) {      $.ajax({          url: '/api/values/' + id,          type: 'DELETE',          contentType: "application/json;charset=utf-8",          success: function (data) {              GetAllBooks();          },          error: function (x, y, z) {              alert(x + '\n' + y + '\n' + z);          }      });  } |

Поскольку это запрос Delete, то он обращается к методу контроллера DeleteBook

Редактирование разбивается на два этапа: загрузка данных в форму редактирования и отправка новых значений на сервер. Для загрузки с сервера данных книги используется функция GetBook:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | function GetBook(id) {        $.ajax({          url: '/api/values/' + id,          type: 'GET',          dataType: 'json',          success: function (data) {              ShowBook(data);          },          error: function (x, y, z) {              alert(x + '\n' + y + '\n' + z);          }      });  } |

Так как это GET-запрос с параметром, то он обращается на контроллере к методу public Book GetBook(int id), а полученную в виде ответа книгу передает для вывода в функцию ShowBook.

Второй этап редактирования - отправка обновленных данных происходит по нажатию кнопки на форме. Так как мы в функции jquery повесили на нее обработчик EditBook. то он будет отправлять новые данные в PUT-запросе (за обработку которого в контроллере отвечает метод EditBook):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | function EditBook() {      var id = $('#editId').val()      // получаем новые значения для редактируемой книги      var book = {          Id: $('#editId').val(),          Name: $('#editName').val(),          Author: $('#editAuthor').val(),          Year: $('#editYear').val()      };      $.ajax({          url: '/api/values/' + id,          type: 'PUT',          data: JSON.stringify(book),          contentType: "application/json;charset=utf-8",          success: function (data) {              GetAllBooks();          },          error: function (x, y, z) {              alert(x + '\n' + y + '\n' + z);          }      });  } |

Аналогично загрузке в PUT-запросе будет действовать создание новой книги и отправка на сервер данных в POST-запросе в функции AddBook.

# Публикация приложения

## Среда публикации и составные части приложения

После того, как приложение написано и протестировано локально, его можно опубликовать, чтобы оно стало доступно широким массам.

Для начала нужно выбрать среду размещения. Мы можем выбрать одну из следующих возможностей:

* **Веб-сервер IIS**
* **Веб-сервер nginx**
* **Облачная платформа Windows Azure**

В зависимости от выбранной среды будут различаться условия конфигурации приложения и его администрирования.

Теперь рассмотрим основные составные части публикуемого сайта, из чего непосредственно будет состоять работающее веб-приложение.

### Статические ресурсы

Статические ресурсы в данном случае - это файлы стилей css, скрипты JavaScript, файлы изображений. В стандартной структуре проекта ASP.NET MVC эти файлы располагаются в каталогах *Content*, *Scripts* и *Images* соответственно. Эти файлы не компилируются в динамические сборки, поэтому являются статическими ресурсами и переносятся на сайт при публикации как есть.

### Сборки приложения

Сборки приложения представляют по сути ключевую часть приложения - именно в них компилируются наши классы и контроллеры. При компиляции сборки приложения попадают в проекте в папку *bin*, где их собственно можно найти. Так, если приложение называется BookStore, то в после компиляции в папке bin окажется файл *BookStore.dll*, который и будет представлять ключевой файл приложения.

При развертывании приложения важно, чтобы оно имело доступ к библиотекам фреймворка ASP.NET MVC. Для этого надо либо установить на сервере данный фреймворк, либо включить все нужные библиотеки фреймворка в пакет приложения. В последнем случае все нужные нам библиотеки также оказываются в папке bin.

В Visual Studio 2012 (в том числе в Express-версии) копирование библиотек в папку bin происходит автоматически). Поэтому все динамические библиотеки фреймворка вы уже сможете увидеть в папке bin после компиляции.

А в Visual Studio 2010 необходимо воспользоваться специальной опцией **Add Deployable Dependencies...**. Для этого нажмем в структуре проекта на название проекта правой кнопкой мыши и в появившемся меню выберем **Add Deployable Dependencies...**.

После этого в появившемся окне выберем **ASP.NET MVC** и нажмем ОК. После этого все необходимые библиотеки фреймворка будут добавлены в папку bin.

### Файлы Global.asax и Web.config

Файлы, содержащие настройки приложения. При развертывании важно учитывать, что некоторые настройки могут меняться в зависимости от сервера. Так, при публикации, возможно, придется менять строку подключения к БД.

### Представления

Представления, которые содержатся в каталоге Views, включается в пакет приложения как есть, в отличие от основной сборки приложения. Затем уже в процессе обращения к приложению в ходе JIT-компиляции представления компилируются и отдаются пользователю в ответ на запрос.

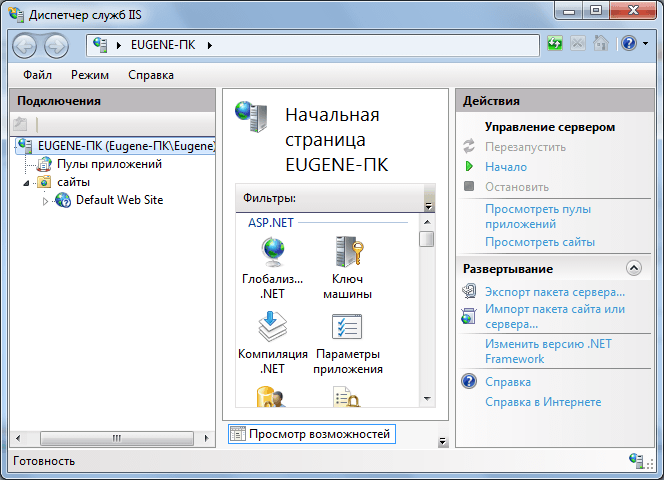
В итоге если у нас такая структура проекта в Visual Studio:

То при публикации (например, в файловой системе) у нас получается примерно такая структура:

## Публикация на веб-сервере IIS

Как правило, большинство веб-приложений ASP.NET публикуются на веб-сервер IIS. Поэтому посмотрим на примере, как размещать веб-приложение на IIS-севере.

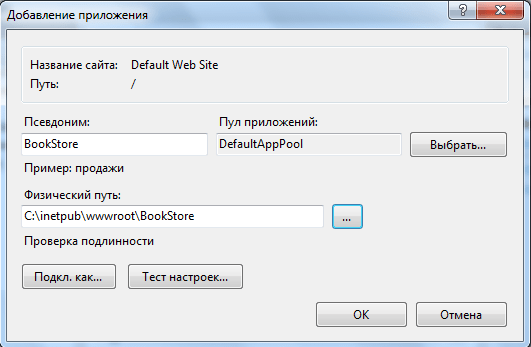
В начале сконфигурируем веб-сервер. Для этого откроем средство администрирования IIS: зайдем в Панель управления, затем выберем Администрирование->Диспетчер служб IIS. И нам откроется консоль управления IIS:



Будем размещать свой сайт в узле по умолчанию (в моем случае это *Default Web Site*). И для этого вначале создадим в каталоге этого узла папку для нашего приложения. По умолчанию каталогом для стандартного веб-узла является каталог *C:\inetpub\wwwroot*. Перейдем в нее и создадим в нем папку BookStore, которая будет содержать наше приложение.

Теперь нажмем правой кнопкой мыши на имя узла по умолчанию и выберем в появившемся меню пункт **Добавить приложение**:

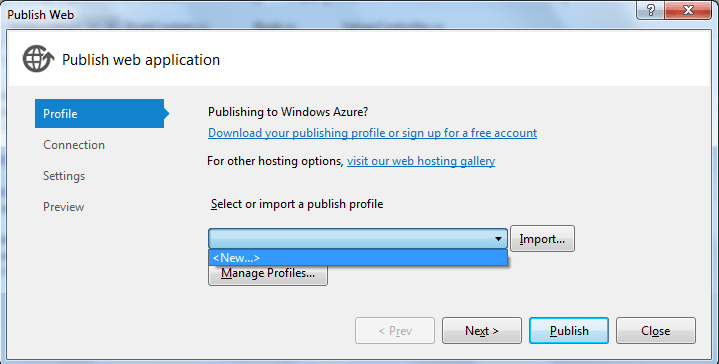
В появившемся окне введем соответствующие настройки (в качестве физического пути приложения созданный выше каталог):



Сайт у нас практически создан. Теперь осталось разместить в каталоге *C:\inetpub\wwwroot\BookStore* наше приложение.

Перейдем к приложению в Visual Studio. Нажмем правой кнопкой на название проекта и в появившемся меню выберем **Publish**:

Перед нами откроется мастер публикации, который предложит нам пройти несколько этапов. В начале выберем профиль:

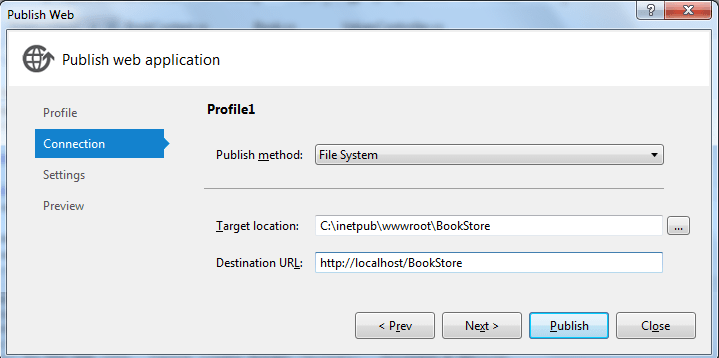


Если не одного профиля не определено, то создадим, нажав на ссылку **New...** и выбрав какое-нибудь название.

После создания профиля нажмем на Next и перейдем к следующему этапу - Connection. На этом этапе для опции **Publish Method** выберем**File System**

Для опции **Target Location** определим физический путь к каталогу нашего сайта. В данном случае это путь *C:\inetpub\wwwroot\BookStore*.

А для поля **Destination URL** указываем url, по которому будет доступно приложение, а именно *http://localhost/BookStore*.



После установки всех свойств жмем на кнопку Publish. После этого в Visual Studio в окне Output студия выдаст сводку об успешности или неуспешности публикации. Возможно, вы увидите сообщения "Отказано в доступе", из-за чего публикация пройдет неудачно:

В этом случае нам надо открыть доступ к папке C:\inetpub\wwwroot\BookStore на запись (по умолчанию стоит доступ только на чтение) и повторить публикацию.

Либо как вариант, если мы не хотим раскрывать доступ на запись, то можно вручную скопировать и перенести все необходимые файлы в указанную выше папку. Как нам показывает окно Output, все файлы для публикации сохраняются в проекте в каталоге*obj\Release\Package\PackageTmp*. И мы можем просто скопировать все файлы из этого каталога и перенести их в каталог*C:\inetpub\wwwroot\BookStore*.

Если мы при создании веб-приложения использовали движок базы данных localdb, то нам надо разместить базу данных на MS SQL Servere и соответсвенно изменить строку подключения к базе данных, чтобы использовать уже не localdb, который предназначен только для целей разработки, а MS SQL Server.

Теперь запустим Службу веб-публикаций через консоль Служб или через консоль IIS и можем обращаться к сайту по url*http://localhost/BookStore*.

## Публикация на внешнем хостинге (на примере somee.com)

Иногда по тем или иным причинам сложно опубликовать или поддерживать сайт на веб-сервере IIS на своем ПК: например, если в наличии нету такого компьютера, который бы можно было использовать в качестве сервера, недостаточная пропускная способность сети, ненужные хлопоты при возможной настройке и конфигурации IIS и т.д. Поэтому часто веб-приложение размещают на каких-то внешних ресурсах, специально предназначенных для хостинга веб-сайтов.

Мы можем выбрать какой-либо хостинг или воспользоваться облачными ресурсами. Одним из таких ресурсов является облачная платформа[Windows Azure](http://www.windowsazure.com/ru-ru/) от компании Microsoft. Использование данной облачной платформы в принципе не бесплатно, причем цены детализированы по многим параметрам, однако можно воспользоваться бесплатным тестовым периодом в течении месяца. На этапе тестового периода, конечно, будут ограничены мощности, но, я думаю, их будет достаточно, чтобы оценить возможности и перспективы своего сайта на данной платформе.

Кроме того, для участников программ MSDN, Microsoft Partner Network и BizSpark предусмотрено бесплатное использование ресурсов данной платформы.

Также мы можем воспользоваться каким-нибудь хостингом с поддержкой ASP.NET. В частности, мы можем воспользоваться хостингом[somee.com](https://somee.com/), который в том числе предлагает и бесплатный хостинг, правда, с ограничениями по сравнению с платными услугами.

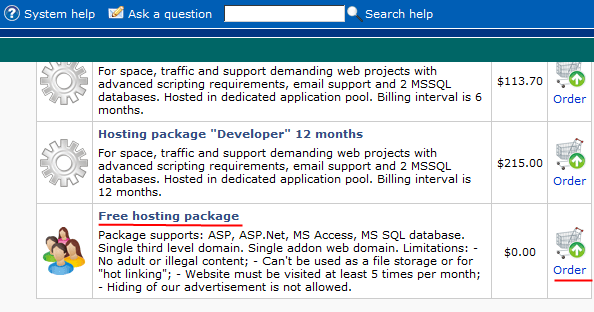
Итак, посмотрим на примере хостинга somee.com, что представляет публикация сайта в сети интернет.

Обращаю внимание, что хостинг somee.com выбран лишь в качестве примера, поскольку предполагает бесплатный хостинг и является очень не плохой площадкой для тестирования приложений asp.net mvc. Работа с другими хостингами может отличаться от приведенных далее указаний.

### Регистрация на somee.com

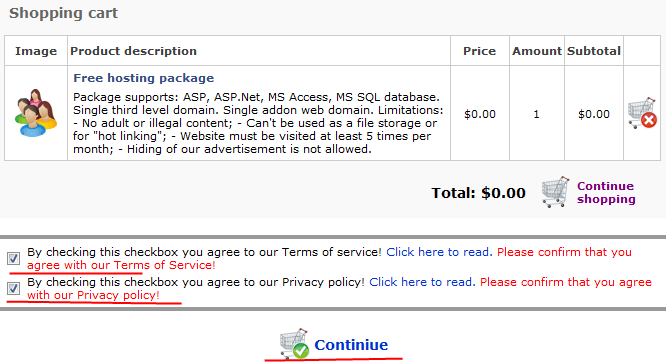
Для начала нам надо зарегистрироваться, создать учетную запись и заказать бесплатный хостинг. Перейдем на главную страницу хостинга[somee.com](https://somee.com/).

В верхнем меню сайта найдем пункт **Cotrol Panel** и нажмем на него. После этого мы окажемся на странице входа учетную запись. Поскольку у нас еще нет учетной записи, нажмем на этой странице на ссылку **Visit our store**. После чего попадем на страницу заказа хостинга. Так как мы заказываем бесплатный тестовый хостинг, то и выберем его - он находится в самом низу списка:

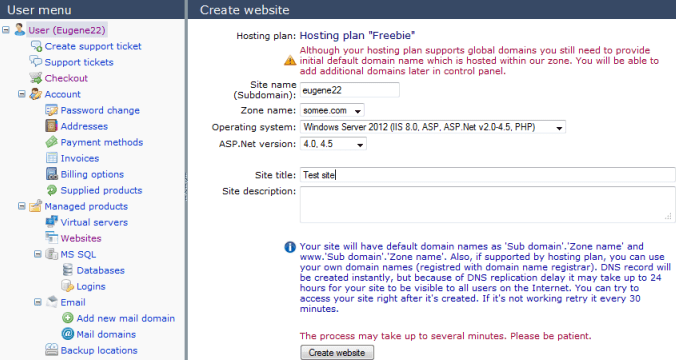


Далее мы попадем на большую форму с регистрацией новой учетной записи. Заполняем все обязательные поля (которые помечены красной звездчкой). Точность вносимой информации никто не проверяет, поэтому можно писать что угодно, главное запомните свои идентификатор пользователя и пароль, под которыми регистрируетесь (поля User ID и Password соответственно), под которыми потом будете входить в систему somee.com:

После заполнения всех обязательных полей нажмем на кнопку внизу формы. И попадем на форму заказа хостинга. Поскольку мы уже заказали бесплатный хостинг, то форма будет содержать соответствующий заказ. Отметим оба флажка на форме и нажмем на кнопку**Continue**:



После чего мы попадем в учетную запись, где надо будет создать сайт. На панели управления учетной записью выберем для сайта какое-нибудь имя и нажмем внизу формы кнопку:

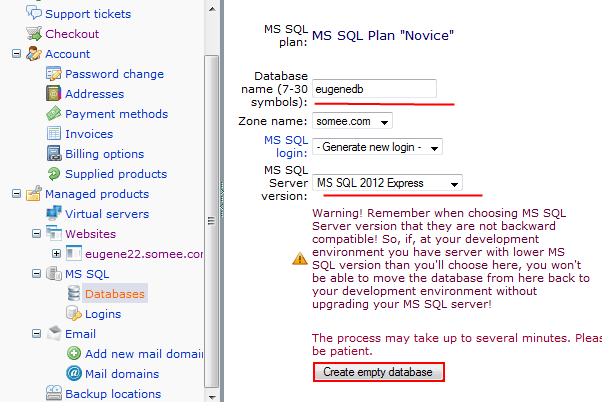


В моем случае имя сайта eugene22, значит, я буду обращаться из браузера к этому сайта через запрос *eugene22.somee.com*.

И если все прошло удачно, сервер известит нас об этом сообщением "You website was created successfully!". После чего мы можем уже загружать на сайт наши файлы приложения и баз данных и управлять сайтом.

## Добавление базы данных на хостинг somee.com

Теперь развернем наш сайт на хостинге. В начале добавим на хостинг базу данных, если она используется. Для этого в панели аккаунта слева находим узел Databases и нажимаем на него. После этого переходим в правой части к созданию новой базы данных. Вводим название для базы данных и выбираем версию SQL-Servera. И после этого нажимаем на кнопку **Create empty database**



Затем нас известят, что создание пустой базы данных прошло успешно, и далее мы можем наполнять ее. Создадим на сервере таблицы по тем таблицам, которые у нас в БД. Во-первых, откроем в Visual Studio определение таблицы бд:

Под визуальным отображением определения таблицы мы увидим скрипт создания таблицы. Например, в моем случае он выглядит так:

CREATE TABLE [dbo].[Books] (

[Id] INT IDENTITY (1, 1) NOT NULL,

[Name] NVARCHAR (50) NOT NULL,

[Author] NVARCHAR (50) NOT NULL,

[Year] INT NOT NULL,

PRIMARY KEY CLUSTERED ([Id] ASC)

);

Теперь найдем на панели учетной записи на somee.com в левом меню пункт **New SQL Query**. Нажмем на него и нам откроется текстовое поле для sql-команды. Скопируем в него скрипт таблицы и нажмем справа на кнопку **Run Single Query**

После этого будет сгенерирована таблица. Затем, если мы хотим перенести на сервер также и данные из таблицы, то отобразим в Visual Studio данные таблицы и по ним сгенерируем скрипт, нажав повыше таблицы на значок Script:

После этого в отдельном окне в Visual Studio будет сгенерирован скрипт добавления данных, который мы также запустим на выполнения с помощью **Run Single Query**. Так, в моем случае скрипт данных будет следующим):

SET IDENTITY\_INSERT [dbo].[Books] ON

INSERT INTO [dbo].[Books] ([Id], [Name], [Author], [Year]) VALUES (1, N'Отцы и дети', N'И. Тургенев', 1862)

INSERT INTO [dbo].[Books] ([Id], [Name], [Author], [Year]) VALUES (2, N'Война и мир', N'Л. Толстой', 1863)

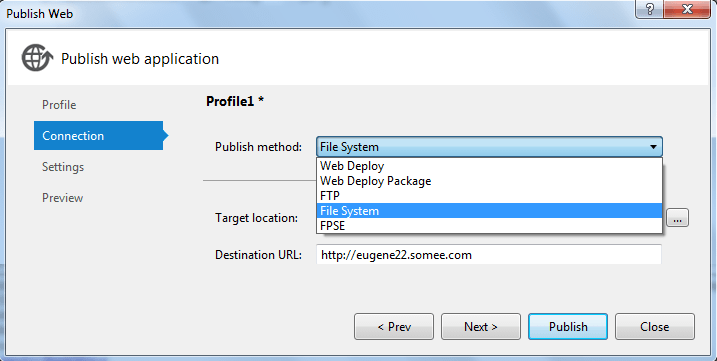
INSERT INTO [dbo].[Books] ([Id], [Name], [Author], [Year]) VALUES (3, N'Преступление и наказание', N'Ф. Достоевский', 1866)

SET IDENTITY\_INSERT [dbo].[Books] OFF

Если таблиц несколько, то повторим данную последовательность действий для всех таблиц.

## Добавление файлов приложения на somee.com

Теперь, когда у нас уже есть база данных на сервере, перейдем к публикации сайта. Нажмем в Visual Studio на проект и выберем в появившемся меню пункт Publish. В мастере публикации выберем профиль (или создадим, если нет ни одного профиля). И перейдем дальше на вкладку Connection (Подключение). Здесь нам надо будет выбрать способ публикации и указать настройки:



Вначале в опции **Publish method** предстоит выбрать способ публикации. Какой способ выбрать в данном случае? В зависимости от ресурса можно использовать различные способы публикации. Так, на Windows Azure можно выбрать способ **Web Deploy**. При наличии доступа по ftp можно выбрать соответствующий способ - **FTP**.

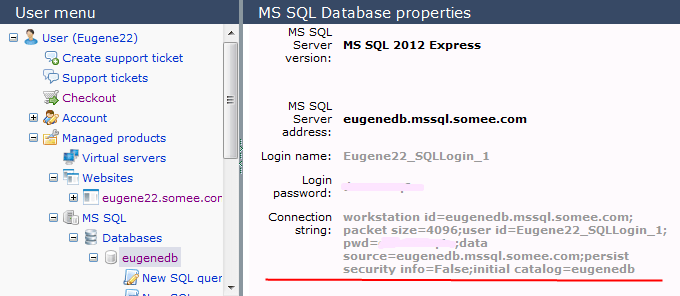
Нам вполне мог бы помочь способ **Web Deploy Package**, только создаваемый на выходе пакет приложения имеет слишком витиеватую структуру и включает ненужные файлы, в частности, каталог App\_Data с бд нам уже не нужен.

Поэтому воспользуемся способом **File System**, который предполагает, что все содержимое сайта будет опубликовано на жестком диске в файловой системе. Затем, выбрав нужные файлы, мы создадим из него архив и перенесем его на хостинг.

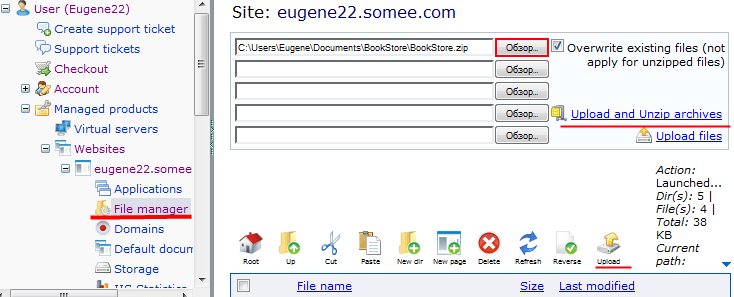
Теперь в поле **Target location** введем каталог на локальном диске, в котором будет опубликовано приложение. А в поле **Destination URL**укажем адрес нашего сайта, который мы создали на somee.com. И нажмем на кнопку Publish.

После этого, если публикация прошла успешно, по указанному пути в поле *Target location* - каталогу публикации - на жестком диске будет опубликовано приложение. Перейдем по этому пути:

Во-первых, изменим строку подключения в файле *web.config* (который у нас в опубликованном каталоге) на новую, поскольку мы создали на хостинге новую бд. Чтобы найти строку подключения к новой базе данных, зайдем в учетную запись на somee.com и слева в меню раскроем узел Databases и нажмем на нашу бд. После этого справа появятся настройки подключения к бд, в том числе и строка подключения:



После изменения строки подключения создадим из всех файлов и папок в каталоге публикации, кроме папки App\_Data, архивный пакет, например, zip. Затем перейдем к панели учетной записи на somee.com и в меню слева выберем File Manager. В центральной части отобразится файловая система нашего сайта. Если в ней есть какие-нибудь файлы по умолчанию, то их можно удалить. Теперь нажмем на кнопку Upload и с помощью кнопки Обзор выберем только что созданный архивный пакет приложения. И после этого нажмем на ссылку**Upload and Unzip archives**.



После этого выбранный пакет приложения будет загружен и автоматически распакован. И теперь мы можем работать с приложением и обращаться к нему из браузера.

Возможно, все шаги по созданию учетной записи и публикации приложения покажутся слишком долгими и скучными, но в реальности это занимает не более 10 минут.

Хостинг somee.com выбран только в качестве примера. Этапы от регистрации до публикации на каждом конкретном хостинге могут отличаться.

# Мобильные приложения на ASP.NET MVC 4

## Разработка мобильных приложений

В настоящее время нас все больше окружают различные мобильные устройства - смартфоны, планшеты, другие гаджеты. И уже не проблема, пребывая к какой-либо точке земного шара, охваченной мобильными сетями, через мобильные устройства выйти в интернет. Уже сейчас, по некоторым оценкам, количество пользователей смартфонов приближается к 1 млрд. человек. Это несет новые возможности по развитию бизнеса, продвижению информационных услуг и т.д. Все это привело развитию веб-сайтов для мобильных устройств.

На заре распространения мобильных телефонов уже существовала веб-разработка для подобных устройств. Например, мы могли по протоколу WAP получать доступ к специальным wap-сайтам, которые были написаны на языке wml:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <?xml version="1.0"?>  <!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN" "<http://www.wapforum.org/DTD/wml_1.1.xml>" >  <wml>    <card id="main" title="WapSite">      <p mode="wrap">Простейшая страница на языке WML.</p>    </card>  </wml> |

Сейчас мобильные телефоны представляют куда большие возможности по получению и отображению содержимого сайтов, а в написании подобных сайтов используется те же HTML/CSS/JavaScript.

Как же создать сайт для мобильных устройств? И чем отличаются сайты для мобильных устройств от традиционных сайтов?

С одной стороны, оба вида сайтов ни чем не отличаются - и там, и там используется разметка html. Более того мы можем создать один единственный сайт для десктопных компьютеров и смартфонов.

Но с другой стороны, размеры мобильных устройств диктуют нам некоторые ограничения, которые мы вынуждены учитывать. В частности, размеры устройств.

Также при разработке обычных сайтов мы нередко сталкиваемся, что те или иные вещи в разных браузерах отображаются по-разному, где-то поддерживается такой стандарт, а где-то нет. С тем же самым нам придется столкнуться и при разработке мобильных сайтов.

### Тестирование мобильных приложений

Для тестирования мобильных приложений конечно же в идеале лучше использовать сами мобильные устройства. Однако мало у кого есть весь спектр мобильных устройств, да и просмотр на смартфоне приложений, размещенных на локальном веб-сервер, было бы проблематично. Поэтому лучше использовать эмуляторы мобильных устройств или мобильных браузеров. Небольшой список подобных эмуляторов можно найти по следующему адресу: <http://www.mobilexweb.com/emulators>.

Лично я предпочитаю использовать эмулятор браузера Opera Mobile, который можно найти по адресу:<http://www.opera.com/ru/developer/mobile-emulator>

### Общие рекомендации по созданию мобильного сайта

Не существует каких-то жестких правил по созданию мобильного сайта, за исключением использования метатега viewport, о котором речь пойдет чуть ниже. Но можно дать несколько рекомендаций общего характера, которые могут помочь при создании мобильного веб-приложения. Например:

* Старайтесь избегать фиксированных размеров, отступов, границ, чтобы общая верстка была в духе так называемого "резинового дизайна"
* Старайтесь не использовать таблицы. Либо желательно, чтобы таблицы имели как можно меньше столбцов (а в идеале вообще один столбец), а их содержимое было кратким. Рекомендуется использовать списки вместо таблиц.
* Избегайте использования фреймов на странице
* Старайтесь использовать небольшие формы с минимальным количеством полей
* Избегайте использования блоков с прокруткой, как горизонтальной, так и вертикальной, поскольку многие устройства испытывают проблемы с рендерингом прокрутки
* Оптимизируйте страницы веб-сайта: сжимайте изображения, оптимизируйте верстку и контент. Избегайте тяжеловесных веб-страниц. Ведь скорость передачи в мобильных сетях в большинстве случаев далека от скорости, например, при проводном подключении. А аппаратные возможности мобильных аппаратов зачастую отстают от возможностей ПК, что будет влиять на скорость обработки веб-страницы.

### Энергоэффективность мобильных сайтов

Говоря о мобильных сайтах, нельзя не затронуть тему энергоэффективности. При создании обычных сайтов нам, как правило, даже в голову не приходит думать об энергоэффективности. Однако, мобильные устройства имеют свои ограничения, и одно из них (особенно на моделях с ОС Андроид) - это запас батареи.

По некоторым подсчетам, парсинг библиотеки jQuery на некоторых устройствах может потреблять до 0,01% заряда полностью заряженной батареи. Возможно, цифра не самая большая, но учитывая, что современные сайты содержат множество нагруженных скриптов, а также стилей, картинок, да и вряд ли пользователи посещают только один сайт. С учетом всего этого вопрос энергоэффективности становится более актуальным.

Как же сделать сайт более энергоэффективным? Во-первых, необходимо использовать только те скрипты, которые действительно нужны веб-сайту. Даже не используемый код JavaScript будет уменьшать заряд батареи.

То же самое относится и к использованию стилей. Загрузка большого количества определения стилей, которое не используется, отнимает также заряда батареи.

Если у вас стоит выбор, что делать: один сайт, для всех устройств с оптимизацией под мобильные устройства, либо два сайта - по одному для ПК и для мобильных устройств, то второй вариант будет выглядеть предпочтительнее. Поскольку в этом случае мобильный аналог сайта будет загружать только предназначенные скрипты и стили, соответственно экономя энергию за счет отсутствия неиспользуемого кода.

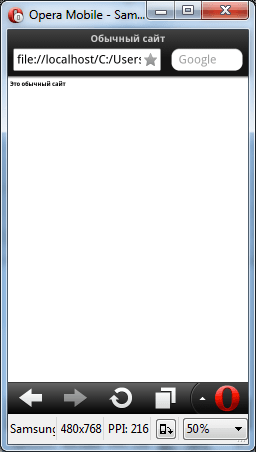
Теперь же перейдем непосредственно к ключевому моменту в разработке мобильных приложений - метатегу viewport.

## Метатег Viewport

Очевидно, что размеры экрана мобильных устройств ставят перед нами одну из наиболее важных проблем при создании сайта для мобильных устройств. В данном случае избежать этой проблемы нам поможет тег Viewport. Возьмем, к примеру, следующую разметку:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <!doctype html>  <html>  <head>      <title>Обычный сайт</title>  </head>  <body>  <h2>Это обычный сайт</h2>  </body>  </html> |

Запустив данную страничку в эмуляторе Opera Mobile, получим следующую картину:



Используя масштабирование, пользователь может наконец-то увидеть, что же там все таки написано. Однако это не очень удобно. При этом сайт страница имеет много пустого места, что не очень красиво.

Почему так происходит? Дело в том, что каждый мобильный браузер задает странице некоторые начальные размеры и потом пытается приспособить под размеры экрана текущего мобильного устройства.

Вся видимая область на экране браузера описывается понятием **Viewport**. По сути viewport представляет область, в которую веб-браузер пытается "впихнуть" веб-страницу. Например, браузер Safari на iPone и iPod определяет ширину viewport по умолчанию равной 980 пикселям. Затем, получив страницу с сервера и вписав в viewport с шириной 980 пикселей, браузер сжимает ее до размеров мобильного устройства. Например, если ширина экрана смартфона составляет 320 пикселей, то до этого размера потом будет сжата страница. И ко всем элементам страницы будет применен коэффициент масштабирования, равный 320/980.

Почему в данном случае используется именно 980 пикселей, а, скажем, не реальный размер экрана? Все дело в том, что по умолчанию браузер считает, что каждая веб-страница предназначена для десктопов. А обычной шириной десктопного сайта можно считать 980 пикселей.

При этом для каждого браузера устанавливается своя ширина области viewport, необязательно 980 пикселей, например, IE поддерживает в качестве начальной ширины 1024 пикселя. Но общая картина будет та же.

Чтобы избежать подобной не очень приятной картины, следует использовать метатег viewport. Использование этого метатега во многом даст понять браузеру, что он имеет дело не с десктопной страницей, а предназначенной в том числе и для мобильных устройств.

Общее объявление метатега следующее:

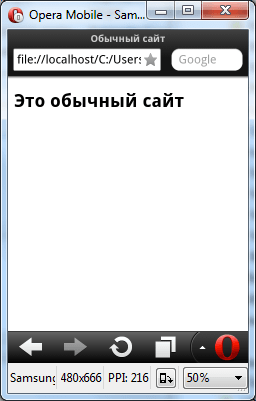
|  |  |
| --- | --- |
| 1 | <meta name="viewport" content="параметры\_метатега"> |

В атрибуте content мета-тега мы можем определить следующие параметры:

|  |  |  |
| --- | --- | --- |
| **Параметр** | **Значения** | **Описание** |
| width | Принимает целочисленное значение в пикселях или значение device-width | Устанавливает ширину области viewport |
| height | Принимает целочисленное значение в пикселях или значение device-height | Устанавливает высоту области viewport |
| initial-scale | Число с плавающей точкой от 0.1 и выше | Задает коэффициент масштабирования начального размера viewport. Значение 1.0 задает отсутствие масштабирования |
| user-scalable | no/yes | Указывает, может ли пользователь с помощью жестов масштабировать страницу |
| minimum-scale | Число с плавающей точкой от 0.1 и выше | Задает минимальный масштаб размера viewport. Значение 1.0 задает отсутствие масштабирования |
| maximum-scale | Число с плавающей точкой от 0.1 и выше | Задает максимальный масштаб размера viewport. Значение 1.0 задает отсутствие масштабирования |

Теперь изменим предыдущий пример веб-страницу, использовав метатег:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | <!doctype html>  <html>  <head>      <title>Обычный сайт</title>      <meta name="viewport" content="width=device-width">  </head>  <body>  <h2>Это обычный сайт</h2>  </body>  </html> |



Веб-страничка определенно выглядит лучше благодаря использованию метатега viewport. Используя параметр width=device-width мы говорим веб-браузеру, что в качестве начальной ширины области viewport надо считать не 980 пикселей или какое-то другое число, а непосредственную ширину экрана устройства. Поэтому затем веб-браузер не будет проводить никакого масштабирования, так как у нас ширина viewport и ширина одинаковы.

Мы также можем использовать другие параметры, например, запретить пользователю масштабировать размеры страницы:

|  |  |
| --- | --- |
| 1 | <meta name="viewport" content="width=device-width, maximum-scale=1.0, minimum-scale=1.0"> |

## Применение CSS в мобильных приложениях

Учитывая особенности мобильных устройств - размеры, поддержку тех или иных стилей CSS и т.д. - может потребоваться определить какой-то особенный стиль для мобильных устройств.

В CSS2 уже было решение в виде правила media, которое позволяет указать устройство, для которого используется данный стиль:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <html>   <head>    <title>Мобильный сайт</title>    <link media="handheld" rel="stylesheet" href="mobile.css">    <link media="screen" rel="stylesheet" href="desktop.css">   </head>   <body>   ...................... |

Правило media="handheld" указывает, что стили в mobile.css будут применяться к мобильным устройствам, в то время как правилоmedia="screen" применяется к десктопным сайтам.

Однако многие современные мобильные браузеры по умолчанию считают, что страница предназначена для десктопов, поэтому в любом случае применяет правило media="screen". Поэтому на подобное решение не стоит полагаться.

Для решения этой проблемы в CSS3 были введен механизм **CSS3 Media Query**. Например, чтобы применить стиль только к мобильным устройствам мы можем написать так:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <html>   <head>    <title>Мобильный сайт</title>    <link rel="stylesheet" type="text/css" media="screen" href="desctop.css" />    <link rel="stylesheet" type="text/css" media="screen and (max-device-width:480px)" href="mobile.css" />   </head>   <body>   ................................ |

Значение атрибута media screen and (max-device-width:480px) говорит нам о том, что стили из файла mobile.css будут применяться к тем устройствам, максимальная ширина экрана которых составляет 480 пикселей - то есть фактически это и есть мобильные устройства.

Как вариант, можно использовать правила CSS3 Media Query в самом файле css:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | @media screen{      body {          background-color: red;      }      /\*Далее остальные стили\*/  }  @media screen and (max-device-width:480px){      body {          background-color: blue;      }      /\*Далее остальные стили\*/  } |

К примеру настройка стилей под смартфоны, планшеты и десктопы может выглядеть так:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | /\*Стили только для смартфонов\*/  @media screen and (max-device-width:480px){      body {          background-color: blue;      }      /\*Далее остальные стили\*/  }  /\*Стили только для планшетов\*/  @media screen and (min-device-width:481px){      body {          background-color: red;      }      /\*Далее остальные стили\*/  }  /\*Стили только для десктопов\*/  @media screen and (min-width:769px){      body {          background-color: yellow;      }      /\*Далее остальные стили\*/  } |

Применяемые функции в CSS3 Media Query:

* **aspect-ratio**: отношение ширины к высоте области отображения (браузера)
* **device-aspect-ratio**: отношение ширины к высоте экрана устройства
* **max-width/min-width и max-height/min-height**: максимальная и минимальная ширина и высота области отображения (браузера)
* **max-device-width/min-device-width и max-device-height/min-device-height**: максимальная и минимальная ширина и высота экрана мобильного устройства
* **orientation**: ориентация (портретная или альбомная)

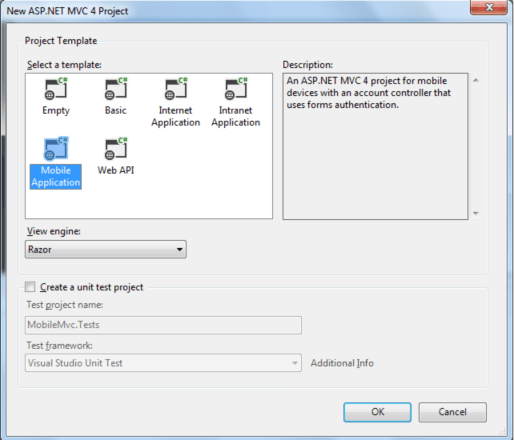
Например, мы можем задать разные стили для разных ориентаций мобильных устройств:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | /\*Стили для портретной ориентации\*/  @media only screen and (orientation: portrait){    }  /\*Стили ландшафтной ориентации\*/  @media only screen and (orientation: landscape){    } |

Таким образом, мы меняем лишь определение стилей в зависимости от устройства, а сами стили css по сути остаются теми же, что мы используем для создания обычных сайтов. Теперь посмотрим, как мы можем создавать мобильные приложения непосредственно на платформе ASP.NET MVC 4.

## Создание мобильного приложения на ASP.NET MVC 4

Итак, создадим первое мобильное приложение на ASP.NET MVC 4. Для этого выберем специально предназначенный для мобильных приложений шаблон **Mobile Application**:



По сути данный тип проекта напоминает проект по типу Internet Application, но только в данном случае он предназначен непосредственно для мобильных устройств.

Одним из ключевых отличий данного проекта будет использование библиотеки jQuery.mobile. Так, в проекте в каталоге Scripts можно найти полную и минимизированную версию библиотеки: *jquery.mobile-1.1.0.js* и *jquery.mobile-1.1.0.min.js*

А в каталоге Content будут находится связанные с jQuery.mobile стили: *jquery.mobile-1.1.0.css* и др.

Собственно наличие библиоетки jQuery.mobile и представляет главное отличие от других типов проектов.

Если мы откроем в проекте мастер-страницу *\_Layout.cshtml*, то мы найдем код, который использует атрибуты jQuery.mobile:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | <!DOCTYPE html>  <html lang="en">      <head>          <meta charset="utf-8" />          <title>@ViewBag.Title</title>          <meta name="viewport" content="width=device-width" />          <link href="~/favicon.ico" rel="shortcut icon" type="image/x-icon" />          @Styles.Render("~/Content/mobileCss", "~/Content/css")          @Scripts.Render("~/bundles/modernizr")      </head>      <body>          <div data-role="page" data-theme="b">              <div data-role="header">                  @if (IsSectionDefined("Header")) {                      @RenderSection("Header")                  } else {                      <h1>@ViewBag.Title</h1>                      @Html.Partial("\_LoginPartial")                  }              </div>              <div data-role="content">                  @RenderBody()              </div>          </div>            @Scripts.Render("~/bundles/jquery", "~/bundles/jquerymobile")          @RenderSection("scripts", required: false)      </body>  </html> |

Чтобы дать понять браузеру, что он имеет дело с сайтом для мобильных устройств, в секции head, идет объявление метатега vieport: <meta name="viewport" content="width=device-width" />.

Далее у нас идут ряд элементов непосредственно с атрибутами jQuery.mobile. Так, тег <div data-role="page" data-theme="b"> определяет роль элемента div на экране - в данном случае элемент играет роль страницы(data-role="page"), а также определяет тему оформления элемента (data-theme="b").

Страница (data-role="page") содержит два элемента div с ролями header (data-role="header") и content (data-role="content"). От роли элемента зависит, как он будет отображаться на экране.

И ниже в секции скриптов идет само подключение библиотеки jQuery.mobile вместе с основной библиотекой jQuery:@Scripts.Render("~/bundles/jquery", "~/bundles/jquerymobile")

Представление *Index.cshtml*, которое будет отображаться при запуске, по сути представляет обычное представление, только опять же использующее арибуты jQuery.mobile:

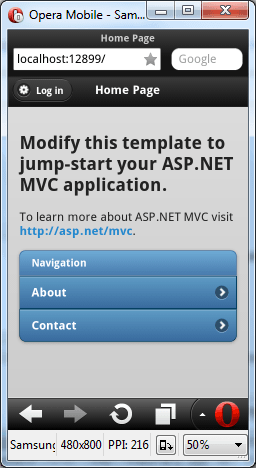
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | @{      ViewBag.Title = "Home Page";  }    <h2>@ViewBag.Message</h2>  <p>      To learn more about ASP.NET MVC visit <a href="<http://asp.net/mvc>" title="ASP.NET MVC Website"><http://asp.net/mvc></a>.  </p>    <ul data-role="listview" data-inset="true">      <li data-role="list-divider">Navigation</li>      <li>@Html.ActionLink("About", "About", "Home")</li>      <li>@Html.ActionLink("Contact", "Contact", "Home")</li>  </ul> |

Использованный здесь атрибут data-role="listview" указывет, что элемент ul будет отображаться в виде списка. Возможно, несовсем ясно, почему мы к списку ul должны применять дополнительный атрибут списка? В данном случае атрибут data-role="listview" лишь просто помогает управлять визуализацией элемента в виде списка на мобильных устройствах.

Нам необязательно использовать библиотеку jquery.mobile, просто она упрощает построение интерфейса для мобильных приложений.

Для более подробного ознакомления с работой библиотеки jQuery.mobile и используемыми ею атрибутами можно обратиться к специализированным источником, а пока в данном случае важно понять суть работы мастер-страницы. То есть, это обычная мастер страницы, только использующая библиотеку jQuery.mobile - ее стили и скрипты, ее атрибуты для управления элементами на странице.

Запустим проект на выполнение и потом обратимся к нему в эмуляторе Opera Mobile:



Теперь, отталкиваясь от этого шаблона и используя библиотеку jQuery.mobile, мы можем создавать мобильные сайты на ASP.NET MVC.

## Режимы отображения DisplayMode

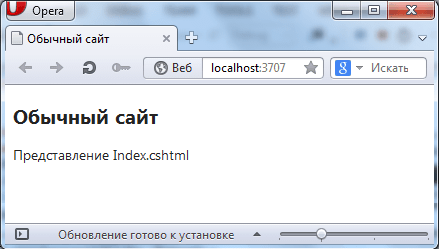
В предыдущем параграфе мы создали мобильное веб-приложение на основе встроенного шаблона. Однако есть еще один способ создания мобильного приложения. Допустим, у нас уже есть проект сайта, но только для десктопов, и нам нужно создать его копию, но только для мобильных приложений.

Инфраструктура ASP.NET MVC 4 содержит такую функциональность как режимы отображения или **DisplayMode**. Например, у нас есть некое представление Index.cshtml. Оно было предназначено для десктопов. И мы хотим рядом с ним сделать копию представления, но только уже для мобильных устройств. Для этого нам достаточно создать в проекте копию данного представления, добавив к его имени суффикс **Mobile**. То есть представление, которое ориентировано на мобильные устройства, у нас будет называться Index.**Mobile**.cshtml.

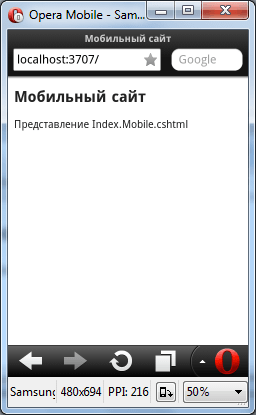
И больше нам ничего не надо менять: контроллеры и их действия остаются теми же. Но теперь этот суффикс Mobile будет показывать фреймворку, что представление *Index.Mobile.cshtml* надо использовать в ответ на запросы, исходящие от мобильных устройств. Во всех остальных случаях будет использоваться обычное представление *Index.cshtml*.

Теперь посмотрим на примере. Возьмем обычный проект asp.net mvc 4 и добавим ко всем его представлениям (в том числе к мастер-странице) двойники с суффиксом Mobile:

Теперь, если мы запустим проект на выполнение и обратимся к приложению из обычного браузера Opera (либо другого браузера, предназначенного для персональных компьютеров), то мы увидим представление Index.cshtml:



В то же время если мы обратимся к приложению из браузера Opera Mobile, то обращение уже будет идти к представлению Index.Mobile.cshtml:



### Переопределение режима DisplayMode

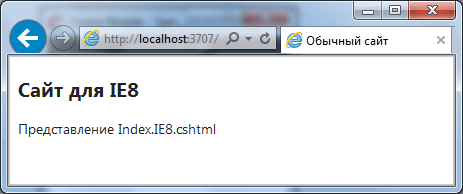
Благодаря суффиксу Mobile фреймворк узнавал о том, что страница предназначена для мобильных устройств. Однако этим работа функциональности DisplayMode не исчерпывается. Так, мы можем переопределить ее действие. Зачем это надо? К примеру ряд браузеров и устройств могут не поддерживать какие-то определенные теги, стили, вследствие этого страницы могут отображаться некорректно.

Так, в настоящее время существует большое число пользователей, которые используют Internet Explorer 8. И мы, допустим, хотим, чтобы наш сайт при запросах от IE 8 использовал только определенные представления - как и в случае с мобильными устройствами.

Для этого вначале нам надо зарегистрировать в файле Global.asax в методе Application\_Start новый режим отображения для Internet Explorer 8:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | using System.Web.WebPages;  ...............          protected void Application\_Start()          {              DisplayModeProvider.Instance.Modes.Insert(0, new DefaultDisplayMode("IE8")              {                  ContextCondition = (context => context.Request.UserAgent.Contains("MSIE 8"))              });          ...................................... |

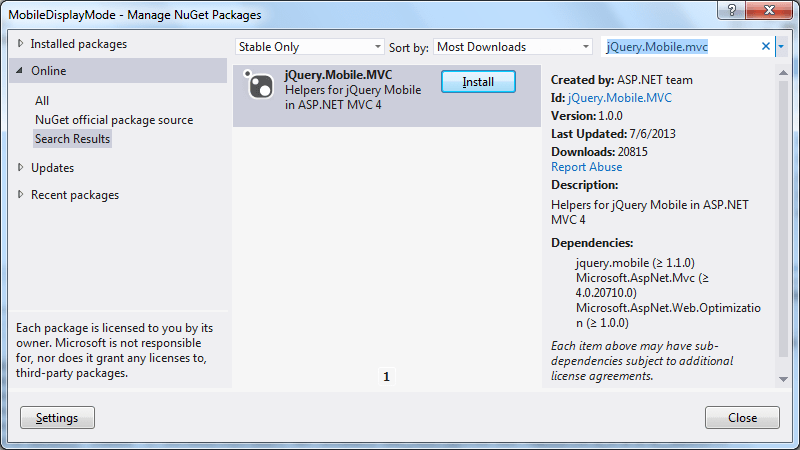
Здесь мы смотрим, содержит ли свойство UserAgent у объекта Request "MSIE 8" - то есть был ли запрос послан браузером IE 8. В этом случае мы используем новый режим DefaultDisplayMode("IE8"). Параметр "IE8" означает, что наши специфичные представления будут иметь названия типа [имя\_представления].IE8.cshtml. Собственно также, как и с мобильными приложениями. Поэтому добавим в проект представление Index.IE8.cshtml и сэмулируем запрос от IE 8, например, в IE 10 в режиме совместимости:



### Переключение между мобильным и десктопным вариантом

Несмотря на то, что автоматическое переключение представлений между мобильной и десктопной версиями очень удобно, все же подобный автоматизм может вызвать неудобства. Десктопные сайты предлагают, как правило, пользователю гораздо больше функционала, чем мобильные. И некоторым пользователям, возможно, больше нравится десктопная версия даже при обзоре на мобильном устройстве.

Чтобы решить эту проблему, ряд сайтов предлагает переключение режимов отображения - с мобильного на десктопный и обратно. Подобную функциональность мы можем использовать и в приложениях asp.net mvc. Для этого вначале добавим в проект с помощью пакетного менеджера NuGet библиотеку **jQuery.Mobile.MVC**:



Этот пакет добавляет в проект ряд файлов:

* Во-первых, добавляется библиотека jquery.mobile.js и ее минимизированная версия, а также используемые этой библиотекой файлы стилей css.
* Во-вторых, добавляется контроллер ViewSwitcherController, который переключает режимы отображения.
* В-третьих, добавляется мастер-страница \_Layout.Mobile.cshtml и файл стилей Site.Mobile.css для мобильного вида (при их отсутствии).
* А также добавляется частичное представление \_ViewSwitcher с разметкой переключения режимов.

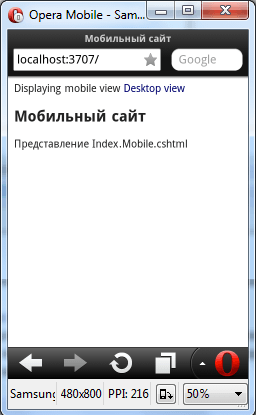
Кроме того, в каталог App\_Start проекта добавляется файл *BundleMobileConfig.cs*, который содержит объявление мобильного бандла, включающего библиотеку jquery.mobile и все сопутствующие файлы стилей. Если мы в будущем собираемся использовать этот бандл, то нам надо зарегистрировать его в файле *Global.asax* в методе App\_Start:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | protected void Application\_Start()  {      BundleMobileConfig.RegisterBundles(BundleTable.Bundles);        //..............................  } |

Теперь, чтобы задействовать функционал переключения, добавим в мастер-страницы layout (и в обычную, и в мобильную версии) вызов частичного представления \_ViewSwitcher (естественно мы можем локализовать представление \_ViewSwitcher.cshtml, настроив нужным нам образом отображения переключения режимов):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <meta name="viewport" content="width=device-width" />      <title>Обычный сайт</title>      @Styles.Render("~/Content/css")      @Scripts.Render("~/bundles/modernizr")  </head>  <body>      @Html.Partial("\_ViewSwitcher")      @RenderBody()        @Scripts.Render("~/bundles/jquery")      @RenderSection("scripts", required: false)  </body>  </html> |

После этого, обратившись к приложению на мобильном устройстве, нам будет доступна функция переключения режимов отображения:



Таким образом, в приложении ASP.NET MVC 4 мы легко можно связать обычную десктопную версию вею-приложения с мобильной и при необходимости осуществлять переключение между ними.

# Бандлы и минификация

## Введение в бандлы и минификацию

В ASP.NET MVC 4 была введена концепция бандлов, которая помогает организовать файлы скриптов и стилей более эффективным путем для снижения издержек при передаче клиенту. Посмотрим, что представляют собой бандлы.

В предыдущих версиях MVC мы могли стандартным образом подключать скрипты и стили, например:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <head>      <meta name="viewport" content="width=device-width" />      <link  type="text/css" rel="stylesheet" href="~/Content/Site.css" />      <script src="~/Scripts/jquery-1.7.1.js" type="text/javascript"></script>      <title>Index</title>  </head> |

Фреймворк MVC для настройки путей предлагает нам использовать URL-хелперы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <head>      <meta name="viewport" content="width=device-width" />      <link  type="text/css" rel="stylesheet" href="@Url.Content("~/Content/Site.css")" />      <script src="@Url.Content("~/Scripts/jquery-1.7.1.js")" type="text/javascript"></script>      <title>Index</title>  </head> |

Использование бандлов представляет совсем другой подход к использованию скриптов и стилей. При создании нового проекта MVC4 по шаблону Basic или Internet Application функциональность бандлов уже по умолчанию включается в приложение. Если вы посмотрите на стандартный код мастер-страницы *\_Layout.cshtml*, то увидите совсем иной способ подключения скриптов и стилей:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <meta name="viewport" content="width=device-width" />      <title>@ViewBag.Title</title>      @Styles.Render("~/Content/css")      @Scripts.Render("~/bundles/modernizr")  </head>  <body>      @RenderBody()        @Scripts.Render("~/bundles/jquery")      @RenderSection("scripts", required: false)  </body>  </html> |

В данном случае подключаются не конкретные скрипты и стили, а именно бандлы. С помощью хелпера Styles.Render подключаются бандлы стилей, а с помощью хелпера Scripts.Render. Эти хелперы принимают в качестве параметра имя бандла.

По умолчанию проекты MVC4 (исключая проекты Empty) уже содержат регистрацию бандлов - в файле *BundleConfig.cs*, который находится в папке *App\_Start*:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | using System.Web;  using System.Web.Optimization;    namespace Guestbook  {      public class BundleConfig      {          // For more information on Bundling, visit <http://go.microsoft.com/fwlink/?LinkId=254725>          public static void RegisterBundles(BundleCollection bundles)          {              bundles.Add(new ScriptBundle("~/bundles/jquery").Include(                          "~/Scripts/jquery-{version}.js"));                bundles.Add(new ScriptBundle("~/bundles/jqueryui").Include(                          "~/Scripts/jquery-ui-{version}.js"));                bundles.Add(new ScriptBundle("~/bundles/jqueryval").Include(                          "~/Scripts/jquery.unobtrusive\*",                          "~/Scripts/jquery.validate\*"));                // Use the development version of Modernizr to develop with and learn from. Then, when you're              // ready for production, use the build tool at [http://modernizr.com](http://modernizr.com/) to pick only the tests you need.              bundles.Add(new ScriptBundle("~/bundles/modernizr").Include(                          "~/Scripts/modernizr-\*"));                bundles.Add(new StyleBundle("~/Content/css").Include("~/Content/site.css"));                bundles.Add(new StyleBundle("~/Content/themes/base/css").Include(                          "~/Content/themes/base/jquery.ui.core.css",                          "~/Content/themes/base/jquery.ui.resizable.css",                          "~/Content/themes/base/jquery.ui.selectable.css",                          "~/Content/themes/base/jquery.ui.accordion.css",                          "~/Content/themes/base/jquery.ui.autocomplete.css",                          "~/Content/themes/base/jquery.ui.button.css",                          "~/Content/themes/base/jquery.ui.dialog.css",                          "~/Content/themes/base/jquery.ui.slider.css",                          "~/Content/themes/base/jquery.ui.tabs.css",                          "~/Content/themes/base/jquery.ui.datepicker.css",                          "~/Content/themes/base/jquery.ui.progressbar.css",                          "~/Content/themes/base/jquery.ui.theme.css"));          }      }  } |

Здесь метод RegisterBundles добавляет все создаваемые бандлы в коллекцию bundles. Объявление бандла выглядит следующим образом:new ScriptBundle("~/bundles/jquery").Include("~/Scripts/jquery-{version}.js").

В конструктор ScriptBundle передается виртуальный путь бандла. А с помощью метода Include уже включаются в данный бандл конкретные файлы скриптов.

В выражении "~/Scripts/jquery-{version}.js" параметр {version} является заменителем для любого символьного обозначения версии скрипта. Это очень удобно, поскольку через некоторое время мы можем поменять версию библиотеки, но при этом в коде нам ничего не придется менять, так как система уже автоматически примет новую версию.

Выражение "~/Scripts/jquery.validate\*" с помощью знака звездочки заменяет остальную часть строки. Например, это выражение подключит в бандл сразу два файла: *jquery.validate.js* и *jquery.validate.unobtrusive.js* (и их минимизированные версии), так как их названия начинаются с jquery.validate\*".

То же самое касается и создания бандлов стилей, только в этом случае используется класс StyleBundle.

Однако само объявление бандлов в файле *BundleConfig.cs* еще не подключает автоматически их в проект. Для этого в файле *Global.asax*прописывается соответствующая директива:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class MvcApplication : System.Web.HttpApplication  {      protected void Application\_Start()      {          //..............................................          BundleConfig.RegisterBundles(BundleTable.Bundles);      }  } |

Таким образом, мы и регистрируем бандлы для нашего приложения.

### Минификация

Ключевым моментом концепции бандлов является минификация. Ее суть состоит в том, что при развертывании приложения клиенту отдается не полная, а минимизированная версия скриптов или стилей. За счет чего экономятся ресурсы сервера, так как идет отдача файла с меньшим объемом. В то же время в процессе отладки приложения мы приложение отдает обычную версию, поскольку благодаря ей, нам проще разобраться в возможных ошибках. Посмотрим на примере. Вот у меня есть некоторая мастер-страничка, которая подключает кучу бандлов:

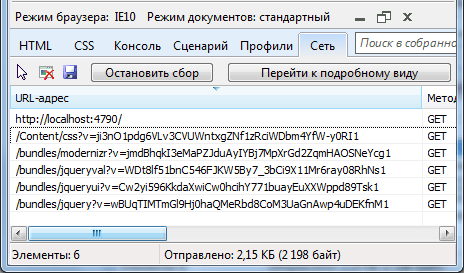
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <meta name="viewport" content="width=device-width" />      <title>@ViewBag.Title</title>      @Styles.Render("~/Content/css")      @Scripts.Render("~/bundles/modernizr")  </head>  <body>      @RenderBody()        @Scripts.Render("~/bundles/jquery")      @Scripts.Render("~/bundles/jqueryui")      @Scripts.Render("~/bundles/jqueryval")      @RenderSection("scripts", required: false)  </body>  </html> |

В данном случае я подключаю пять бандлов. Если запустить теперь приложение, то через средства разработчика в IE10 я могу увидеть скрипты, которые несут бандлы:

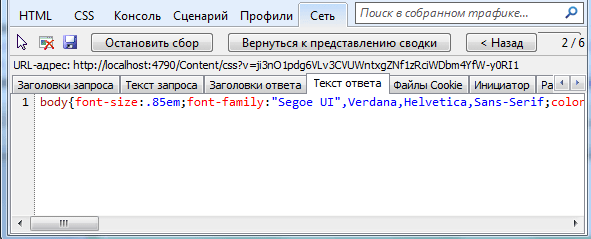
В данном случае у нас по умолчанию приложение находится в режиме отладки. В файле *web.config* можно найти следующую секцию:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <configuration>    <!--------------------------------------->    <system.web>      <httpRuntime targetFramework="4.5" />      <compilation debug="true" targetFramework="4.5" />    <!---------------------------------------> |

Параметр debug="true" указывает, что приложение запускается в режиме отладки. Теперь изменим его значение на false: debug="false" и перезапустим проект. Теперь мы увидим несколько иной результат:



И если вы посмотрите, например, в том же IE10 в средствах разработчика заголовки ответа, то вы увидите, что в этот раз в режиме развертывания сервер отдает нам минимизированный файл:



Таким образом, использование бандлов в проектах MVC4 повышает эффективность веб-приложения.

Кроме того, для минимизации скриптов и стилей существует большое количество утилит, которые можно дополнительно использовать. Например, очень неплохая утилита по минимизации скриптов [Microsoft Ajax Minifier](http://ajaxmin.codeplex.com/), позволяющая ужать скрипты в среднем до 50%, а иногда и больше.

## Создание бандлов. Библиотеки из CDN

Создадим свой бандл. Например, я определил в папке со скриптами подкаталог MyScripts для каких-то своих скриптов, которые я хочу подключать в своем бандле:

В файле *BundleConfig.cs* я добавляю свой бандл с моими файлами:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | using System.Web;  using System.Web.Optimization;    namespace Guestbook  {      public class BundleConfig      {          public static void RegisterBundles(BundleCollection bundles)          {                bundles.Add(new ScriptBundle("~/bundles/myscripts").Include(                          "~/Scripts/MyScripts/\*.js"));                bundles.Add(new ScriptBundle("~/bundles/jquery").Include(                          "~/Scripts/jquery-{version}.js"));                  // Далее идет подключение остальных бандлов          }      }  } |

Выражение "~/Scripts/MyScripts/\*.js" подключит в бандл все файлы js, которые есть в каталоге Scripts/MyScripts. Далее переходим к разметке и подключаем там новый бандл:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <meta name="viewport" content="width=device-width" />      <title>@ViewBag.Title</title>      @Styles.Render("~/Content/css")      @Scripts.Render("~/bundles/modernizr")  </head>  <body>      @RenderBody()        @Scripts.Render("~/bundles/jquery")      @Scripts.Render("~/bundles/myscripts")      @RenderSection("scripts", required: false)  </body>  </html> |

И после этого бандл будет подключаться в приложение. При подключении бандлов не стоит забывать об очередности: если у вас есть бандлы, файлы которых зависят от других файлов в других бандлов, например, от jquery, то, естественно, зависимые бандлы надо подключать после основных.

### Подключение библиотек из сетей CDN

Обычным делом на сегодняшний день является подключение различных популярных библиотек javascript из сетей CDN. Если мы хотим подключать, например, библиотеку jquery не со своего приложения, а с какой-нибудь сети CDN, то мы должны включить поддержку CDN в бандлах в файле *BundleConfig.cs*:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | using System.Web;  using System.Web.Optimization;    namespace Guestbook  {      public class BundleConfig      {          public static void RegisterBundles(BundleCollection bundles)          {              // Оригинальное подключение закомментируем, так как будем подключаться из сети CDN              //bundles.Add(new ScriptBundle("~/bundles/jquery").Include(              //            "~/Scripts/jquery-{version}.js"));                bundles.UseCdn = true;   //включаем поддержку CDN                // добавляем адрес нужной библиотеки в какой-нибудь сети CDN              var jqueryCdnPath = "<http://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.7.1.min.js>";                bundles.Add(new ScriptBundle("~/bundles/jquery",                          jqueryCdnPath).Include(                          "~/Scripts/jquery-{version}.js"));                // Далее остальные бандлы          }      }  } |

Далее подключаем бандл обычным способом. Кроме того, мы можем задать возможность подключения библиотеки из своего приложения, если библиотеку не удалось загрузить с этой сети:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <meta name="viewport" content="width=device-width" />      <title>@ViewBag.Title</title>      @Styles.Render("~/Content/css")      @Scripts.Render("~/bundles/modernizr")  </head>  <body>      @RenderBody()        @Scripts.Render("~/bundles/jquery")      <script type="text/javascript">              if (typeof jQuery == 'undefined') {                  var e = document.createElement('script');                  e.src = '@Url.Content("~/Scripts/jquery-1.7.1.js")';                  e.type = 'text/javascript';                  document.getElementsByTagName("head")[0].appendChild(e);              }      </script>      @RenderSection("scripts", required: false)  </body>  </html> |

# Мультиязычный сайт и локализация

## Введение в мультиязычные сайты. Ресурсы

Сегодня никого не удивишь мультиязычными сайтами. Благодаря мультиязычночти сайт становится доступным гораздо большему числу потенциальных пользователей, то сулит большие выгоды, например, увеличение дохода, увеличение аудитории и т.д. Посмотрим, как в ASP.NET MVC 4 мы можем создать мультиязычный сайт.

Перед созданием мультиязычного сайта надо определиться с концепцией.

* Мы можем сделать для каждой отдельной культуры свое представление и затем в зависимости от выбранной культуры рендерить его в ответ клиенту.
* Второй вариант предполагает использование ресурсов, когда у нас одно представление для всех культур, и в зависимости от выбранной культуры подгружаются определенные для нее ресурсы.

Используем второй вариант. Создадим проект по типу Basic (в моем случае он будет называться MultilingualSite). И первым делом создадим модель. Допустим, наше приложение будет работать со списком пользователей. Добавим в папку Models модель **Person**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | using System;  using System.ComponentModel.DataAnnotations;    namespace MultilingualSite.Models  {      public class Person      {          public int Id { get; set; }            [Required(ErrorMessageResourceType = typeof(Resources.Resource),                    ErrorMessageResourceName = "NameRequired")]          [Display(Name = "Name", ResourceType = typeof(Resources.Resource))]          public string Name { get; set; }            [Required(ErrorMessageResourceType = typeof(Resources.Resource),                    ErrorMessageResourceName = "CountryRequired")]          [Display(Name = "Country", ResourceType = typeof(Resources.Resource))]          public string State { get; set; }      }  } |

В данном случае для упрощения примера я использовал всего три поля. К двоим применены атрибуты Required и Display. Только в данном случае атрибут Required указывает на тип ресурса, который будет применяться (**ErrorMessageResourceType**) и имя ресурса (**ErrorMessageResourceName**).

Определение ErrorMessageResourceType = typeof(Resources.Resource), ErrorMessageResourceName = "CountryRequired" по сути говорит о том, что у нас есть некоторое пространство имен ресурсов Resources, в нем определен тип Resource. А имя ресурса представляет уже конкретный ресурс, значение которого будет использоваться в качестве сообщения об ошибке при валидации. То же самое можно прописать и для других атрибутов валидации, например, для атрибута StringLength.

Пока у нас нет никаких ресурсов, но сейчас мы их создадим.

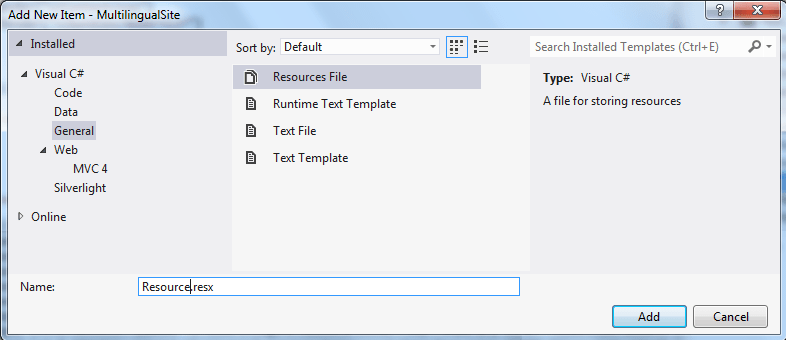
То же самое характерно и для атрибута Display, только в данном случае его свойство Name указывает на имя ресурса, который будет использоваться в качестве названия для соответствующего поля.

Также создадим какой-нибудь простенький контекст данных PersonContext:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | sing System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.Data.Entity;    namespace MultilingualSite.Models  {      public class PersonContext : DbContext      {          public DbSet<Person> Persons { get; set; }      }  } |

Теперь создадим ресурсы. Ресурсы могут содержать не только сообщения об ошибках валидации, но и другие данные, например, заголовки страниц и т.д.

Создадим в проекте папку *Resources*. В эту папку добавим файл ресурсов *Resource.resx*. Это будет файл ресурсов по умолчанию, который будет использоваться для русского языка:



Перед использованием ресурсов укажем некоторые их настройки для каждого ресурса в окне Свойств (Propertires) внизу под окном проекта:

Здесь нам надо установить следующие параметры:

* **Build Action** - в качестве типа построения выберем значение **Embedded Resource**
* **Custom Tool** - в качестве инструмента создания ресурсов введем в поле **PublicResXFileCodeGenerator**
* **Custom Tool Namespace** - в качестве пространства имен укажем **Resources**. В данном случае важно указать именно то пространство имен, которое мы собираемся использовать. Используя выше атрибуты при определении модели, мы указалиErrorMessageResourceType = typeof(Resources.Resource), где в качестве пространства имен выступает Resources.

Теперь перейдем к заполнению самих ресурсов. Нажмем на файл *Resource.resx* и в открывшуюся таблицу внесем названия ресурсов и их значения:

Это и есть весь набор используемых ресурсов, и мы можем определить множество разных ресурсов. Подобный набор надо указать и у других ресурсов, только указав значения на соответствующем языке. Так, на рисунке указан ресурс NameRequired. При определении модели в атрибуте я указал, что именно этот ресурс будет использоваться в качестве сообщения об ошибке валидации.

Кроме того, мы можем в коде представлений обратиться к ресурсам, например: @Resources.Resource.NameRequired. Здесь Resources - пространство имен, Resource - тип ресурса (образуется от названия файла ресурсов), а NameRequired - имя. В таком же формате можно использовать и другие ресурсы.

Теперь добавим ресурсы для других культур. Для этого можно скопировать файл Resource.resx и вставить его копии в папку Resources. Я сделал две копии файла и переименовал их следующим образом:

* Resource.en.resx : файл ресурсов для английского языка
* Resource.de.resx : файл ресурсов для немецкого языка

Можно ограничиться и одним, например, только для английского языка.

Теперь структура проекта выглядит так:

### Условности при создании ресурсов

При локализации имеют значение в наименовании ресурсов суффиксы, обозначающие локали. Так, мы можем использовать нейтральные культуры, например: Resource.en.resx. Здесь суффикс en обозначает английский язык в целом, все англоязычные культуры. Но мы можем конкретизировать культуру, например, задать ресурс только для американской версии английского: Resource.en-US.resx.

Суффикс культуры имеет следующий формат: languagecode-country/regioncode, где languagecode - код языка, а country/regioncode - код субкультуры. В моем случае я использую только нейтральные культуры без субкультур.

Откроем файл Resource.en.resx и переведем все ресурсы на английский язык:

С ресурсами мы завершили, теперь перейдем к построению самой логики приложения.

## Логика мультиязычного сайта. Фильтр локализации

В предыдущем параграфе мы создали проект, добавив в него модель и ресурсы, которые затем будем использовать при локализации. Теперь создадим минималистичную логику приложения.

Существуют различные способы определения культуры и непосредственной локализации, например, можно переопределять класс контроллера, доопределяя в нем OnActionExecuted и ExecuteCore. Но в данном случае мы поступим по-другому и создадим свой фильтр действий, который будет срабатывать при обращении к действиям контроллера и производить локализацию.

Создадим в проекте папку *Filters* и добавим в нее класс *CultureAttribute* со следующим содержанием:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | using System;  using System.Collections.Generic;  using System.Globalization;  using System.Threading;  using System.Web;  using System.Web.Mvc;    namespace MultilingualSite.Filters  {      public class CultureAttribute : FilterAttribute, IActionFilter      {          public void OnActionExecuted(ActionExecutedContext filterContext)          {              string cultureName = null;              // Получаем куки из контекста, которые могут содержать установленную культуру              HttpCookie cultureCookie = filterContext.HttpContext.Request.Cookies["lang"];              if (cultureCookie != null)                  cultureName = cultureCookie.Value;              else                  cultureName = "ru";                // Список культур              List<string> cultures = new List<string>() {"ru", "en", "de"};              if (!cultures.Contains(cultureName))              {                  cultureName = "ru";              }                  Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture(cultureName);                  Thread.CurrentThread.CurrentUICulture = CultureInfo.CreateSpecificCulture(cultureName);          }            public void OnActionExecuting(ActionExecutingContext filterContext)          {              //не реализован          }      }  } |

Метод OnActionExecuted срабатывает после вызова действия контроллера. В начале он получает установленную культуру из куков. Затем мы смотрим, есть ли такая культура в списке. Обратите внимание, что строковые значения культур в списке соответствуют суффиксам культур в названиях файлов ресурсов (за исключением русского языка, для которого используется культура по умолчанию).

Сама же установка культуры производится в следующих строках:

|  |  |
| --- | --- |
| 1  2 | Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture(cultureName);  Thread.CurrentThread.CurrentUICulture = CultureInfo.CreateSpecificCulture(cultureName); |

После этого для локализации система будет выбирать нужный файл ресурсов.

Теперь создадим собственно само приложение: контроллер и представления. Добавим в папку Controllers контроллер *HomeController*:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46 | using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.Web.Mvc;  using MultilingualSite.Filters;  using MultilingualSite.Models;    namespace MultilingualSite.Controllers  {      [Culture]      public class HomeController : Controller      {          PersonContext db = new PersonContext();            public ActionResult Index()          {              return View(db.Persons);          }            public ActionResult ChangeCulture(string lang)          {              string returnUrl = Request.UrlReferrer.AbsolutePath;              // Список культур              List<string> cultures = new List<string>() {"ru", "en", "de"};              if (!cultures.Contains(lang))              {                  lang = "ru";              }              // Сохраняем выбранную культуру в куки              HttpCookie cookie = Request.Cookies["lang"];              if (cookie != null)                  cookie.Value = lang;   // если куки уже установлено, то обновляем значение              else              {                    cookie = new HttpCookie("lang");                  cookie.HttpOnly = false;                  cookie.Value = lang;                  cookie.Expires = DateTime.Now.AddYears(1);              }              Response.Cookies.Add(cookie);              return Redirect(returnUrl);          }      }  } |

По сути контроллер от обычного отличается только тем, что к нему применен ранее созданный атрибут. И практически всю логику по локализации выполняет этот атрибут.

Также в контроллере определено действие ChangeCulture, которое меняет язык с помощью установки нужных куков и затем перенаправляет на ту же страницу.

Теперь добавим представление *Index.cshtml*, которое будет выглядеть так:

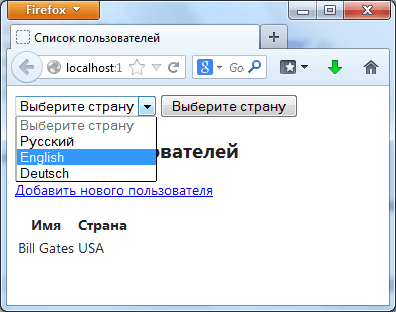
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | @model IEnumerable<MultilingualSite.Models.Person>    @{      ViewBag.Title = @Resources.Resource.IndexHeader;  }    <h2>@Resources.Resource.IndexHeader</h2>  <p>      @Html.ActionLink(@Resources.Resource.CreateLink, "Create")  </p>  <table>      <tr>          <th>              @Html.DisplayNameFor(model => model.Name)          </th>          <th>              @Html.DisplayNameFor(model => model.State)          </th>      </tr>    @foreach (var item in Model) {      <tr>          <td>              @Html.DisplayFor(modelItem => item.Name)          </td>          <td>              @Html.DisplayFor(modelItem => item.State)          </td>      </tr>  }  </table> |

Это обычное представление, только вместо прямых вставок текста используются ресурсы: @Resources.Resource.IndexHeader. И изменим стандартную мастер-страницу *\_Layout.cshtml*, добавив на нее функционал переключения языков:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <meta name="viewport" content="width=device-width" />      <title>@ViewBag.Title</title>      @Styles.Render("~/Content/css")      @Scripts.Render("~/bundles/modernizr")  </head>  <body>      <div>          @using (Html.BeginForm("ChangeCulture", "Home"))          {              <select name="lang">                  <option disabled selected>@Resources.Resource.ChooseLang</option>                  <option value="ru">Русский</option>                  <option value="en">English</option>                  <option value="de">Deutsch</option>             </select>              <input type="submit" name="submit" value="@Resources.Resource.ChooseLang">          }      </div>      @RenderBody()        @Scripts.Render("~/bundles/jquery")      @RenderSection("scripts", required: false)  </body>  </html> |

Здесь мы добавили простую форму с выпадающим списком, которая будет отсылать выбор пользователя на сервер вышеопределенному методу ChangeCulture.

Запустим, и у нас отобразится страница на русском языке:



Выбрав в выпадающем списке другой язык и нажав Enter, мы получим локализацию уже на другой язык.

Также мы можем создать и другие представления, например, для создания пользователей. Но общие принципы останутся те же.

# Работа с SignalR

## Введение в SignalR

В последнее время в веб-среде все чаще создаются веб-приложения, использующие технологии коммуникаций в реальном времени: это и простые чаты, и более сложные многопользовательские видеоконференции. Такие приложения могут использовать различные приемы работы: технологию Web Socket, опросы long polling и т.д. Для упрощения работы с коммуникациями реального времени была создана специальная библиотека под названием **SignalR**.

При этом SignalR не является библиотекой, предназначенной только для использования в ASP.NET MVC. Ее также можно использовать и в веб-формах, а также в консольных приложениях, десктопных приложения и в WPF.

SignalR предоставляет простой API для создания функционала, который позволяет вызывать функции JavaScript на стороне клиента из серверного кода, написанного с помощью языков платформы .NET. SignalR значительно упрощает работу с коммуникациями реального времени. Библиотека обрабатывает все подключения и автоматически рассылает сообщения всем подключенным клиентам, либо каким-нибудь специфическим клиентам.

Фактически библиотека SignalR состоит из API серверной стороны, который применяется в коде на C#, и из клиентских библиотек JavaScript.

SignalR предоставляет разработчикам две модели: постоянные подключения (**Persistent Connection**) и хабы (**Hubs**).

Постоянные подключения (Persistent Connection API) представляют разработчикам прямой доступ к низкоуровневому протоколу коммуникации. Подключения в этой модели представляют конечную точку, к которой подключаются клиенты, наподобие модели подключений в WCF.

Хабы же предоставляют протокол взаимодействия более высокого уровня. Они представляют верхний слой над Persistent Connection API и позволяют клиенту и серверу напрямую вызывать методы друг друга.

Если вы будете работать с SignalR, то скорее всего будете преимущественно использовать хабы, как более удобные инструменты.

#### Поддерживаемые платформы

SignalR поддерживается наиболее распространенными серверными платформами на базе ОС Windows:

* **Windows Server 2012**
* **Windows Server 2008 r2**
* **Windows 8**
* **Windows 7**
* **Windows Azure**

Для работы на стороне сервера также необходима версия фреймворка .net от 4.0 и выше.

Поддерживаемые клиентские платформы (браузеры) тоже многообразны:

* Microsoft Internet Explorer 8, 9 и 10. Версии Modern, Desktop и Mobile
* Mozilla Firefox: как на ОС Windows, так и на Mac ОС
* Google Chrome: как на ОС Windows, так и на Mac ОС
* Safari: как на ОС Windows, так и на Mac ОС
* Opera: ОС Windows
* Android-браузер

Также следует учитывать, что браузер должен поддерживать библиотеку jQuery 1.6.4 или более поздние версии (например, 1.7.2, 1.8.2, 1.9.1 и т.д.).

#### Транспорт передачи данных

Для обмена данными между клиентом и сервером SignalR использует тот способ передачи или тот транспорт, который наиболее подходит к данной ситуации. Однако разработчики могут переопределить способ передачи. SignalR предоставляет следующие типы технологий для взаимодействия сервера и клиента:

* **WebSockets**
* **Server-sent events**
* **Forever Frames**
* **Long polling**

#### Выбор транспорта

При создании подключения SignalR выбирает, если доступно, технологию WebSocket, так как это наиболее оптимальная технология для SignalR, наиболее эффективно использующая память сервера. В то же время WebSocket может использоваться только на серверах под управлением Windows Server 2012 или Windows 8 и при наличии установленного .NET Framework 4.5. При этом также технология WebSocket должна поддерживаться и браузером клиента. И если технология WebSocket недоступна на сервере или клиенте, то выбирается другой транспорт.

Если WebSocket, то SignalR использует технологию Server Sent Events, при ее поддержке сервером и клиентом.

Если технология Server Sent Events также недоступна, то используются скрытые фреймы - Forever Frames.

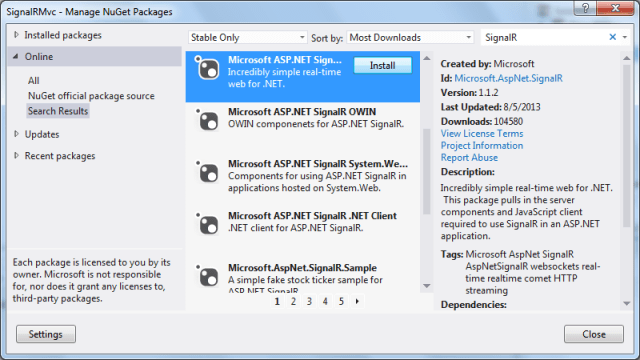
И если Forever Frames также недоступны, то используется Long Polling. Например, если на стороне клиента используется браузер IE 8 и ниже, то используется Long Polling.

Теперь рассмотрим работу с библиотекой на конкретных примерах.

## Первое приложение с SignalR

Создадим простенькое приложение с использованием SignalR - небольшой чат. Итак, создадим новое приложение ASP.NET MVC4 по шаблону Basic. Я назвал свое приложение **SignalRMvc**.

После создания проекта найдем через пакетный менеджер NuGet библиотеку **Microsoft ASP.NET SignalR**:



Установим ее. После этого в папку библиотек *Referenses* будет добавлен ряд библиотек SignalR, а в каталог скриптов *Scripts* будет добавлен клиентский скрипт *jquery.signalR-[номер\_версии].js* и его минимизированный аналог.

Поскольку приложение чата оперирует пользователями, то создадим модель пользователей. Добавим в папку *Models* класс *User*:

|  |  |
| --- | --- |
| 1  2  3  4  5 | public class User  {      public string ConnectionId { get; set; }      public string Name { get; set; }  } |

Затем в папку *Controllers* добавим пустой контроллер *HomeController*, где будет один метод Index:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | using System;  using System.Web;  using System.Web.Mvc;    namespace SignalRMvc.Controllers  {      public class HomeController : Controller      {          public ActionResult Index()          {              return View();          }      }  } |

Больше нам ничего не надо, всю логику приложения будет содержать хаб.

Теперь добавим в проект новую папку. Назовем ее *Hubs*. В ней будут находиться хабы нашего приложения. В эту папку добавим новый класс *ChatHub.cs*. Он будет иметь следующий код:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52 | using System;  using System.Web;  using System.Collections.Generic;  using System.Linq;  using Microsoft.AspNet.SignalR;  using SignalRMvc.Models;    namespace SignalRChat  {      public class ChatHub : Hub      {          static List<User> Users = new List<User>();            // Отправка сообщений          public void Send(string name, string message)          {              Clients.All.addMessage(name, message);          }            // Подключение нового пользователя          public void Connect(string userName)          {              var id = Context.ConnectionId;                  if (Users.Count(x => x.ConnectionId == id) == 0)              {                  Users.Add(new User { ConnectionId = id, Name = userName });                    // Посылаем сообщение текущему пользователю                  Clients.Caller.onConnected(id, userName, Users);                    // Посылаем сообщение всем пользователям, кроме текущего                  Clients.AllExcept(id).onNewUserConnected(id, userName);              }          }            // Отключение пользователя          public override System.Threading.Tasks.Task OnDisconnected()          {              var item = Users.FirstOrDefault(x => x.ConnectionId == Context.ConnectionId);              if (item != null)              {                  Users.Remove(item);                  var id = Context.ConnectionId;                  Clients.All.onUserDisconnected(id, item.Name);              }                return base.OnDisconnected();          }      }  } |

Как уже ранее говорилось, SignalR использует две модели взаимодействия сервера и клиента: Persistent Connection и хабы. В данном случае мы используем хабы. Для этого создаем свой хаб ChatHub, который наследуется от класса Hub.

Во-первых, мы создаем список, который будет хранить подключенных к чату пользователей.

Далее у нас определен ряд методов для отправки сообщений, подключения и отключения пользователей. Разберем метод Send, который предназначен В нем вызывается единственный метод: Clients.All.addMessage(name, message);. Объект Clients означает коллекцию всех пользователей хаба. Свойство All, идущее далее, говорит о том, что метод надо применить у всех подключенных клиентов.

#### Формат вызова методов клиента

* Вызов метода на всех клиентах: Clients.All.addMessage(name, message);
* Вызов метода только на текущем клиенте, который обратился к серверу: Clients.Caller.addMessage(name, message);
* Вызов метода на всех клиентах, кроме того, который обратился к серверу: Clients.Others.addMessage(name, message);
* Вызов метода только у клиента с определенным id: Clients.Client(Context.ConnectionId).addMessage(name, message);
* Вызов метода на всех клиентах, кроме клиента с определенным id: Clients.AllExcept(connectionId).addMessage(name, message);
* Вызов метода на всех клиентах указанной группы: Clients.Group(groupName).addMessage(name, message);
* Вызов метода на всех клиентах указанной группы, за исключением клиента, у которого id - connectionId: Clients.Group(groupName, connectionId).addMessage(name, message);
* Вызов метода на всех клиентах указанной группы, за исключением обратившегося к серверу клиента:Clients.OthersInGroup(groupName).addMessage(name, message);

В зависимости от того, кому надо передать сообщение, мы можем выбрать один из вариантов.

Далее в выражении следует метод addMessage. Этот метод объявляется на стороне клиента в коде javascript. Чуть позже мы добавим код клиентской стороны. А пока просто надо знать, что эти методы находятся не на серверной части, а на стороне клиента.

В методе Connect мы сначала получаем id текущего пользователя, который и обратился к методу Connect, через объектContext.ConnectionId. Этот id задается средой и хранит строковое значение (не числовое). Затем вызываем методы на клиенте через объектClients.

В конце переопределяем метод OnDisconnected, который выполняет отключение текущего клиента в асинхронном режиме. Например, при закрытии вкладки браузера клиент по сути выходит из приложения и вызывается метод OnDisconnected для текущего клиента.

Затем для метода Index ранее созданного контроллера Home создадим представление *Index.cshtml*:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117 | @{      ViewBag.Title = "Чат";  }    <h2>Чат-комната</h2>    <div class="main">      <div id="loginBlock">          Введите логин:<br />          <input id="txtUserName" type="text" />          <br />          <input id="btnLogin" type="button" value="Войти" />      </div>      <div id="chatBody">          <div id="header"></div>          <div id="inputForm">              <input type="text" id="message" />              <input type="button" id="sendmessage" value="Отправить" />          </div>          <div id="chatroom">              <ul></ul>          </div>            <div id="chatusers">              <p><strong>Все пользователи</strong></p>              <ul></ul>          </div>      </div>      <input id="hdId" type="hidden" />      <input id="username" type="hidden" />  </div>  @section scripts {      <!--Ссылка на библиотеку SignalR -->      <script src="~/Scripts/jquery.signalR-1.1.2.min.js"></script>      <!--Ссылка на автоматически сгенерированный скрипт хаба SignalR -->      <script src="~/signalr/hubs"></script>      <script>          $(function () {                $('#loginBlock').show();              $('#chatBody').hide();              // Ссылка на автоматически-сгенерированный прокси хаба              var chat = $.connection.chatHub;              // Объявление функции, которая хаб вызывает при получении сообщений              chat.client.addMessage = function (name, message) {                  // Добавление сообщений на веб-страницу                  $('#chatroom ul').append('<li><strong>' + htmlEncode(name)                      + '</strong>: ' + htmlEncode(message) + '</li>');              };                // Функция, вызываемая при подключении нового пользователя              chat.client.onConnected = function (id, userName, allUsers) {                    $('#loginBlock').hide();                  $('#chatBody').show();                  // установка в скрытых полях имени и id текущего пользователя                  $('#hdId').val(id);                  $('#username').val(userName);                  $('#header').html('<h3>Добро пожаловать, ' + userName + '</h3>');                    // Добавление всех пользователей                  for (i = 0; i < allUsers.length; i++) {                        AddUser(allUsers[i].ConnectionId, allUsers[i].Name);                  }              }                // Добавляем нового пользователя              chat.client.onNewUserConnected = function (id, name) {                    AddUser(id, name);              }                // Удаляем пользователя              chat.client.onUserDisconnected = function (id, userName) {                    $('#' + id).remove();              }                // Открываем соединение              $.connection.hub.start().done(function () {                    $('#sendmessage').click(function () {                      // Вызываем у хаба метод Send                      chat.server.send($('#username').val(), $('#message').val());                      $('#message').val('');                  });                    // обработка логина                  $("#btnLogin").click(function () {                        var name = $("#txtUserName").val();                      if (name.length > 0) {                          chat.server.connect(name);                      }                      else {                          alert("Введите имя");                      }                  });              });          });          // Кодирование тегов          function htmlEncode(value) {              var encodedValue = $('<div />').text(value).html();              return encodedValue;          }          //Добавление нового пользователя          function AddUser(id, name) {                var userId = $('#hdId').val();              if (userId != id){                    $("#chatusers ul").append('<li id="' + id + '" class="user" >' + name + '</li>');              }          }      </script>  } |

Разметка содержит по сути два блока: loginBlock (блок ввода логина) и chatBody (сам чат). В один момент времени виде только один блок, поэтому, если пользователь удачно подключился, то мы переключаем видимость, делая видимым блок chatBody.

Блок chatBody содержит три подблока: inputForm (форма ввода сообщения), chatroom (поле, куда будут добавляться сообщения) и chatusers (список всех пользователей, кроме текущего).

Теперь перейдем к коду клиентской части. Обратите внимание, что весь javascript у нас находится в специальной секции скриптов, которая подключается в мастер-страницу после библиотеки jQuery. Если у вас несколько другая структура приложения, то перед данным скриптом следует подключать основную библиотеку jQuery.

Вначале подключаем библиотеку SignalR. (Также обратите внимание, что новейшая версия библиотеки может отличаться от использованной здесь.) После этого идет подключение скрипта хаба, который будет автоматически сгенерирован при запуске приложения.

Весь код находится в функции jQuery, за ее пределами определены две функции - htmlEncode (для кодирования тегов, чтобы пресечь возможные попытки вставок кода javascript и прочие нехорошести) и AddUser (для добавления данных пользователя в список).

Чтобы взаимодействовать с хабом, получаем прокси хаба: var chat = $.connection.chatHub;. Затем определяем ряд функций клиентаchat.client.addMessage, chat.client.onConnected и т.д.

Выше в хабе в коде C# у нас было определено обращение к функциям клиента: Clients.All.addMessage(name, message);. Функция addMessage - это и есть функция, определенная для chat.client.addMessage. Подобным образом мы можем обращаться на сервере и к другим функциям клиента.

Для открытия подключения мы вызываем метод $.connection.hub.start().done(), передавая в него функцию. В этой функции мы вешаем обработчики кнопок, по нажатию на которые происходит обращение на сервер.

Например, отправку сообщения производится с помощью вызова chat.server.send($('#username').val(), $('#message').val());. Выражениеchat.server представляет собой обращение к методам хаба на сервере. То есть в данном случае будет идти обращение к методу public void Send(string name, string message), определенному в классе ChatHub.

Подобным образом срабатывает вызов метода chat.server.connect(name);. Обратите внимание, что хотя в коде c# методы объявлены с большой буквы, в коде javascript в их названии используется малая буква.

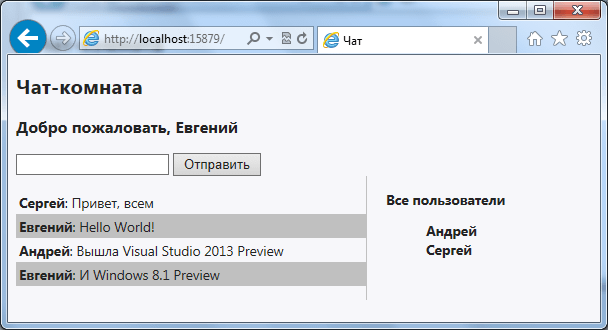
И в заключении нам надо зарегистрировать хаб. Для этого откроем файл *Global.asax* и добавим в метод Application\_Start строкуRouteTable.Routes.MapHubs(), которая и производит регистрацию:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | protected void Application\_Start()  {      RouteTable.Routes.MapHubs();      AreaRegistration.RegisterAllAreas();        WebApiConfig.Register(GlobalConfiguration.Configuration);      FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);      RouteConfig.RegisterRoutes(RouteTable.Routes);      BundleConfig.RegisterBundles(BundleTable.Bundles);  } |

Ну также можно применить некоторые стили для общей красоты. Свои стили я приводить здесь не буду, чтобы не перегружать пример.

Теперь мы можем проверить в действии. После запуска нам будет предложено ввести логин:

После успешного логина форма ввода логина станет невидимой, зато станет доступна форма самого чата. Можно открыть несколько вкладок в браузере с данным приложением, подключив тем самым несколько клиентов, и протестировать:



Таким образом, слева у нас список с сообщениями, а справа - список всех пользователей, кроме текущего.

## Persistent Connection API

Persistent Connection API предлагает нам низкоуровневый интерфейс для создания приложений. Возможно, многим он покажется менее удобным, чем использование хабов, но тоже может использоваться как вариант построения приложения. Итак, создадим проект приложения, которое будет использовать Persistent Connection.

Создадим проект ASP.NET MVC 4 по типу Basic. Назовем его, к примеру, *PersistentConMvc*. Затем через менеджер NuGet добавим в проект библиотеку SignalR.

Добавим стандартный контроллер в папку *Controllers*. Пусть это будет HomeController с одним действием Index:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | using System;  using System.Web;  using System.Web.Mvc;    namespace PersistentConMvc.Controllers  {      public class HomeController : Controller      {          public ActionResult Index()          {              return View();          }      }  } |

Затем добавим в папку *Models* модель Data. Эта модель будет представлять объект передаваемого сообщения. В принципе мы могли бы обмениваться простыми строковыми объектами. Однако использование более сложных объектов позволит нам структурировать передаваемые данные, легче ими управлять. Итак, модель Data будет иметь следующий код:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | using System;    namespace PersistentConMvc.Models  {      public class Data      {          public string Name { get; set; } // Имя пользователя          public string Message { get; set; } // Сообщение пользователя      }  } |

Этот объект будет передаваться от пользователя серверу.

Теперь создадим серверную инфраструктуру для взаимодействия с клиентом. Создадим в проекте папку *Connection* и добавим в нее новый класс *ChatConnection*:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | using System;  using System.Collections.Generic;  using System.Web;  using Microsoft.AspNet.SignalR;  using PersistentConMvc.Models;  using System.Threading.Tasks;  using Newtonsoft.Json;    namespace PersistentConMvc.Connection  {      public class ChatConnection : PersistentConnection      {          protected override Task OnConnected(IRequest request, string connectionId)          {              Data chatData = new Data() { Name = "Сообщение сервера", Message = "Пользователь "+connectionId+" присоединился к чату" };              return Connection.Broadcast(chatData);          }            protected override Task OnReceived(IRequest request, string connectionId, string data)          {              Data chatData = JsonConvert.DeserializeObject<Data>(data);              return Connection.Broadcast(chatData);          }            protected override Task OnDisconnected(IRequest request, string connectionId)          {              Data chatData = new Data() { Name = "Сообщение сервера", Message = "Пользователь " + connectionId + " покинул чат" };              return Connection.Broadcast(chatData);          }      }  } |

Класс ChatConnection наследуется от класса PersistentConnection и переопределяет некоторые его методы.

Метод OnConnected срабатывает при каждом подключении клиента. В него передаются два параметра: IRequest request, который представляет объект запроса, и connectionId, который представляет id, автоматически присваемый обратившемуся к серверу клиенту.

Здесь мы просто посылаем клиенту в методе Connection.Broadcast объект Data. Метод Broadcast может принимать любой объект, в том числе и простую строку, главное, чтобы в коде клиента мы могли распарсить должным образом это сообщение.

Метод OnDisconnected аналогичен методу OnConnected, только выполняется при отключении клиента.

При получении сообщения от клиента срабатывает метод OnReceived: полученное сообщение передается в третьем параметре data. Поскольку это сообщение представляет строку, мы его десериализуем в объект Data и затем рассылаем его всем подключенным клиентам.

Для того, чтобы подключение и использование Persistent Connection API было доступно, зарегистрируем его в файле *Global.asax* в методеApplication\_Start:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | protected void Application\_Start()  {        RouteTable.Routes.MapConnection<ChatConnection>("chat", "/chat");        AreaRegistration.RegisterAllAreas();        WebApiConfig.Register(GlobalConfiguration.Configuration);      FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);      RouteConfig.RegisterRoutes(RouteTable.Routes);      BundleConfig.RegisterBundles(BundleTable.Bundles);  } |

Строка RouteTable.Routes.MapConnection<ChatConnection>("chat", "/chat"); регистрирует маршрут, по которому мы будем обращаться из клиентской части.

Теперь клиентская часть. Создадим для метода Index контроллера Home представление *Index.cshtml*:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73 | @{      ViewBag.Title = "Чат";  }    <h2>Чат</h2>        <div id="loginBlock">           Введите логин:<br />           <input id="txtUserName" type="text" /><br />           <input id="btnLogin" type="button" value="Войти" />      </div>      <div id="chatBody">          <div id="header"></div>          <div id="inputForm">              <input type="text" id="message" />              <input type="button" id="sendmessage" value="Отправить" />          </div>          <div id="chatroom">              <ul></ul>          </div>        </div>  <input id="username" type="hidden" />    @section scripts {      <!--Ссылка на библиотеку SignalR -->      <script src="~/Scripts/jquery.signalR-1.1.3.min.js"></script>      <script>      $(function () {            $('#chatBody').hide();          // получаем соединение          var myConnection = $.connection("/chat");          // обработка получения данных от сервера          myConnection.received(function (data) {              $("#chatroom ul").append("<li><strong>" + htmlEncode(data.Name) +                  '</strong>: ' + htmlEncode(data.Message) + "</li>");          });                // обработка логина              $("#btnLogin").click(function () {                    var userName = $("#txtUserName").val().replace(/\s/g, '');                  if (userName.length > 0) {                      $('#username').val(userName);                        //отключаем поля ввода логина                      $('#btnLogin').attr('disabled', 'disabled');                      $('#txtUserName').attr('disabled', 'disabled');                        $('#chatBody').show();                      // открываем соединение                      myConnection.start().done(function () {                          //обработчик отправки сообщения                          $('#sendmessage').click(function () {                             // посылаем сериализованный объект на сервер                              myConnection.send(JSON.stringify({ name: $('#username').val(), message: $('#message').val() }));                              $('#message').val('');                          });                      });                  }                  else {                      alert("Введите имя");                  }              });       });      // Кодирование тегов      function htmlEncode(value) {          var encodedValue = $('<div />').text(value).html();          return encodedValue;      }      </script>  } |

Итак, разметка html содержит два блока: блок ввода ника loginBlock и блок самого чата chatBody, в котором есть форма ввода сообщения и список для полученных сообщений.

В коде javascript первым делом мы создаем подключение: var myConnection = $.connection("/chat");. В качестве параметра указывается тот маршрут, который мы выше зарегистрировали в файле Global.asax.

Функция myConnection.received вызывается при получении клиентом от сервера данных. В данном случае поскольку сервер передает клиенту объект Data, у которого есть свойства Name и Message, то мы можем их получить в данной функции и добавить на страницу в список сообщений.

Далее определена основная логика. Ее суть: после ввода ника пользователем открывается соединение с сервером, и пользователь может отправлять сообщения.

Сначала мы получаем введенный ник, убираем из него все пробелы, и если он проходит проверку, добавляем его в скрытое поле, из которого ник потом будет добавляться к отсылаемому на сервер сообщению.

Чтобы пользователь не мог второй раз ввести ник и подключиться, отключаем поля ввода и наконец открываем соединение (myConnection.start().done(...).

Для отправления на сервер используется метод myConnection.send, в который в качестве параметра передаются введенное сообщение и сохраненный в скрытом поле ник. С помощью функции JSON.stringify данные превращаются в объект json, который потом сервер извлекает в методе protected override Task OnReceived(IRequest request, string connectionId, string data) из последнего параметра.

По сути небольшой примитивный чат готов, мы можем применить к нему стили по своему вкусу и протестировать:

