|  |
| --- |
| , RD Dep. |

ASP.NET.010 MVC

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| REVISION HISTORY | | | | | |
| Ver. | Description of Change | Author | Date | Approved | |
| Name | Effective Date |
| <1.0> | Первая версия | Крамар Юлия | <1.04.2015> |  |  |
|  |  |  |  |  |  |

СОДЕРЖАНИЕ

[1. Введение в ASP.NET MVC 4](#_Toc430916327)

[1.1. Особенности платформы ASP.NET MVC 4](#_Toc430916328)

[1.2. Начало работы с ASP.NET MVC 4 5](#_Toc430916329)

[1.3. Структура проекта MVC 4 6](#_Toc430916330)

[2. Контроллеры 6](#_Toc430916331)

[2.1. Основы контроллеров 6](#_Toc430916332)

[2.2. Методы действий 9](#_Toc430916333)

[2.3. Получение входных данных 10](#_Toc430916334)

[2.4. Результаты действий 11](#_Toc430916335)

[2.5. ViewResult 14](#_Toc430916336)

[2.6. Переадресация и отправка кодов статуса и ошибок 15](#_Toc430916337)

[2.7. Отправка файлов в ASP.NET MVC 17](#_Toc430916338)

[2.8. Передача данных из контроллера в представление 18](#_Toc430916339)

[2.9. ControllerContext и HttpContext. 21](#_Toc430916340)

[2.10. Асинхронные контроллеры и методы 23](#_Toc430916341)

[3. Представления и модели 25](#_Toc430916342)

[3.1. Введение в представления 25](#_Toc430916343)

[3.2. Строго типизированные представления 29](#_Toc430916344)

[3.3. Мастер-страницы 31](#_Toc430916345)

[3.4. Частичные представления 34](#_Toc430916346)

[3.5. Работа с движком представления Razor 37](#_Toc430916347)

[3.6. Понятие движка представления 38](#_Toc430916348)

[3.7. HTML-хелперы 45](#_Toc430916349)

[3.8. Работа с формами 47](#_Toc430916350)

[3.9. Строго типизированные хелперы 52](#_Toc430916351)

[3.10. Шаблонные хелперы 55](#_Toc430916352)

[3.11. Редактирование модели 57](#_Toc430916353)

[3.12. Добавление модели 59](#_Toc430916354)

[3.13. Удаление модели 60](#_Toc430916355)

[3.14. Модели со сложной структурой 62](#_Toc430916356)

[3.15. Передача массивов и сложных данных в контроллер 67](#_Toc430916357)

[4. Метаданные и валидация модели 70](#_Toc430916358)

[4.1. Аннотации данных для отображения свойств 70](#_Toc430916359)

[4.2. Основы валидации 75](#_Toc430916360)

[4.3. Атрибуты валидации 78](#_Toc430916361)

[4.4. Валидация модели в контроллере 82](#_Toc430916362)

[4.5. Отображение ошибок валидации 84](#_Toc430916363)

[4.6. Создание собственной логики валидации 87](#_Toc430916364)

[5. Привязка модели 93](#_Toc430916365)

[5.1. Введение в привязку моделей 93](#_Toc430916366)

[5.1.1. Привязчики моделей 93](#_Toc430916367)

[5.2. DefaultModelBinder 94](#_Toc430916368)

[5.2.1. Выборочная привязка 95](#_Toc430916369)

[6. Авторизация и аутентификация в MVC 4 96](#_Toc430916370)

[6.1. Введение в авторизацию и аутентификацию 96](#_Toc430916371)

[6.2. Аутентификация Windows 98](#_Toc430916372)

[6.3. Аутентификация форм 101](#_Toc430916373)

[6.4. Авторизация в MVC 4 и SimpleMembershipProvider 105](#_Toc430916374)

[6.5. Использование универсальных провайдеров 108](#_Toc430916375)

[6.6. Web Site Administration Tool 114](#_Toc430916376)

# Введение в ASP.NET MVC

## Особенности платформы ASP.NET MVC

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.

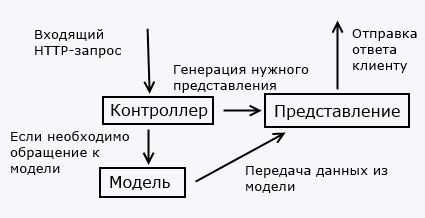
Шаблон MVC, лежащий в основе новой платформы, подразумевает взаимодействие трех компонентов: контроллера (controller), модели (model) и представления (view).

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 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.

## Начало работы с ASP.NET MVC 4

Чтобы создавать приложения на ASP.NET MVC потребуется среда разработки - Microsoft Visual Studio Express 2012 for Web (либо другой выпуск Visual Studio 2010/2012/2013, например, Visual Web Developer 2010 Express. Если вы используете Visual Web Developer 2010 Express, а не 2012-ю версию, то надо доустановить также средства разработки для MVC 3/4, так как данный выпуск по умолчанию идет с MVC 2. Далее мы будем ориентироваться преимущественно на MVC 4. Visual Studio 2012 уже включает средства разработки для MVC 4.

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.

В дополнение к этим проектам в обновлениях Visual Studio 2012 и в Visual Studio 2013 были добавлены еще два типа проектов:

* **Single Page Application**. Шаблон, предназначенный для создания сайтов-одностраничников.
* **Facebook Application**. Предназначен для создания веб-приложений под социальную сеть Facebook.

Кроме того при создании приложения необходимо 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).

## Структура проекта MVC 4

#### 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

# Контроллеры

## Основы контроллеров

Контроллер является центральным компонентом в архитектуре MVC. Он 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);  } |

Создание простого контроллера, реализующего данный интерфейс (вручную):

|  |
| --- |
| 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();          }      }  } |

**Члены класса Controller**

|  |  |
| --- | --- |
| **Имя** | **Описание** |
| [ActionInvoker](https://msdn.microsoft.com/ru-ru/library/system.web.mvc.controller.actioninvoker%28v=vs.118%29.aspx) | Получает средство вызова действий для контроллера. |
| [AsyncManager](https://msdn.microsoft.com/ru-ru/library/system.web.mvc.controller.asyncmanager%28v=vs.118%29.aspx) | Обеспечивает асинхронные операции. |
| [Binders](https://msdn.microsoft.com/ru-ru/library/system.web.mvc.controller.binders%28v=vs.118%29.aspx) | Получает или задает связыватель. |
| [ControllerContext](https://msdn.microsoft.com/ru-ru/library/system.web.mvc.controllerbase.controllercontext%28v=vs.118%29.aspx) | Возвращает или задает контекст контроллера. (Производный от [ControllerBase](https://msdn.microsoft.com/ru-ru/library/system.web.mvc.controllerbase%28v=vs.118%29.aspx).) |
| [DisableAsyncSupport](https://msdn.microsoft.com/ru-ru/library/system.web.mvc.controller.disableasyncsupport%28v=vs.118%29.aspx) | Определяет, следует ли отключить поддержку асинхронных операций для контроллера. |
| [HttpContext](https://msdn.microsoft.com/ru-ru/library/system.web.mvc.controller.httpcontext%28v=vs.118%29.aspx) | Получает связанные с НТТР сведения об индивидуальном НТТР-запросе. |
| [ModelState](https://msdn.microsoft.com/ru-ru/library/system.web.mvc.controller.modelstate%28v=vs.118%29.aspx) | Получает объект словаря состояния модели, содержащий состояние модели и проверку привязки модели. |
| [Profile](https://msdn.microsoft.com/ru-ru/library/system.web.mvc.controller.profile%28v=vs.118%29.aspx) | Получает профиль HTTP-контекста. |
| [Request](https://msdn.microsoft.com/ru-ru/library/system.web.mvc.controller.request%28v=vs.118%29.aspx) | Получает объект HttpRequestBase для текущего HTTP-запроса. |
| [Response](https://msdn.microsoft.com/ru-ru/library/system.web.mvc.controller.response%28v=vs.118%29.aspx) | Получает объект HttpResponseBase для текущего HTTP-ответа. |
| [RouteData](https://msdn.microsoft.com/ru-ru/library/system.web.mvc.controller.routedata%28v=vs.118%29.aspx) | Получает данные маршрута для текущего запроса. |
| [Server](https://msdn.microsoft.com/ru-ru/library/system.web.mvc.controller.server%28v=vs.118%29.aspx) | Получает объект HttpServerUtilityBase, предоставляющий методы, используемые при обработке веб-запросов. |
| [Session](https://msdn.microsoft.com/ru-ru/library/system.web.mvc.controller.session%28v=vs.118%29.aspx) | Получает объект HttpSessionStateBase для текущего HTTP-запроса. |
| [TempData](https://msdn.microsoft.com/ru-ru/library/system.web.mvc.controllerbase.tempdata%28v=vs.118%29.aspx) | Получает или задает словарь для временных данных. (Производный от [ControllerBase](https://msdn.microsoft.com/ru-ru/library/system.web.mvc.controllerbase%28v=vs.118%29.aspx).) |
| [TempDataProvider](https://msdn.microsoft.com/ru-ru/library/system.web.mvc.controller.tempdataprovider%28v=vs.118%29.aspx) | Получает объект поставщика временных данных, используемый для хранения данных для следующего запроса. |
| [Url](https://msdn.microsoft.com/ru-ru/library/system.web.mvc.controller.url%28v=vs.118%29.aspx) | Получает объект вспомогательного метода URL-адреса, используемый для создания URL-адресов с помощью маршрутизации. |
| [User](https://msdn.microsoft.com/ru-ru/library/system.web.mvc.controller.user%28v=vs.118%29.aspx) | Получает сведения о безопасности пользователя для текущего HTTP-запроса. |
| [ValidateRequest](https://msdn.microsoft.com/ru-ru/library/system.web.mvc.controllerbase.validaterequest%28v=vs.118%29.aspx) | Получает или задает значение, указывающее, включена ли проверка запросов для данного запроса. (Производный от [ControllerBase](https://msdn.microsoft.com/ru-ru/library/system.web.mvc.controllerbase%28v=vs.118%29.aspx).) |
| [ValueProvider](https://msdn.microsoft.com/ru-ru/library/system.web.mvc.controllerbase.valueprovider%28v=vs.118%29.aspx) | Получает или задает поставщик значений для контроллера. (Производный от [ControllerBase](https://msdn.microsoft.com/ru-ru/library/system.web.mvc.controllerbase%28v=vs.118%29.aspx).) |
| [ViewBag](https://msdn.microsoft.com/ru-ru/library/system.web.mvc.controllerbase.viewbag%28v=vs.118%29.aspx) | Получает или динамический словарь данных представления. (Производный от [ControllerBase](https://msdn.microsoft.com/ru-ru/library/system.web.mvc.controllerbase%28v=vs.118%29.aspx).) |
| [ViewData](https://msdn.microsoft.com/ru-ru/library/system.web.mvc.controllerbase.viewdata%28v=vs.118%29.aspx) | Получает или задает словарь для данных представления. (Производный от [ControllerBase](https://msdn.microsoft.com/ru-ru/library/system.web.mvc.controllerbase%28v=vs.118%29.aspx).) |
| [ViewEngineCollection](https://msdn.microsoft.com/ru-ru/library/system.web.mvc.controller.viewenginecollection%28v=vs.118%29.aspx) | Получает коллекцию обработчиков представлений. |

## Методы действий

Методы действий (action methods) представляют такие методы контроллера, которые обрабатывают запросы по определенному URL. Например, рассмотрим контроллер:

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

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

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

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

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

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

## Получение входных данных

В рассмотренном примере 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 в примере с вычислением площади треугольника возвращался 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 свой тип результатов действий. Он 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);

Используем класс для создания результата в методе дейстия контроллера:

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

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

Создадим еще один класс результатов 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 набор классов результатов действий, которые охватывают большинство, если не все возможные ситуации.

* **ContentResult**: пишет указанный контент напрямую в ответ в виде строки

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");      }  } |

Если такого пути не окажется, то приложение выбросит ошибку.

## 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 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:

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

Подобным образом мы можем послать браузеру любой другой статусный код.

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

|  |
| --- |
| public ActionResult Check(int age)  {      if (age < 21)      {          return HttpNotFound();      }      return View();  } |

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

|  |
| --- |
| public ActionResult Check(int age)  {      if (age < 21)      {          return new HttpUnauthorizedResult();      }      return View();  } |

## 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. И в зависимости от выбранного способа используется соответствующая перегруженная версия этого метода.

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

|  |
| --- |
| public FileResult GetFile()  {      // Путь к файлу      string file\_path = Server.MapPath("~/Files/PDFIcon.pdf");      // Тип файла - content-type      string file\_type="application/pdf";      // Имя файла - необязательно      string file\_name = "PDFIcon.pdf";      return File(file\_path,file\_type,file\_name);  } |

Предполагается существование папки 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 класс FileContentResult, только вместо имени файла в методе File указывается массив байтов, в который был считан файл:

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

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

|  |
| --- |
| // Отправка потока  public FileResult GetStream()  {      string path = Server.MapPath("~/Files/PDFIcon.pdf");      // Объект Stream      FileStream fs = new FileStream(path, FileMode.Open);      string file\_type = "application/pdf";      string file\_name = "PDFIcon.pdf";      return File(fs, file\_type, file\_name);  } |

## 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, можно добавить 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.

Характеристики ViewData

1. ViewData —словарный объект, производный от TempViewDataDictionary;
2. используется для передачи данных из контроллера в соответствующее представление;
3. жизненный цикл ограничен текущим запросом;
4. если происходит redirect, значение ViewData превращается в null;
5. необходимо осуществлять приведение типов и проверять на null, чтобы избежать ошибок;

|  |
| --- |
| // запись во viewData  public ActionResult Index()  {  var personInfo = context.Personinfos.FirstOrDefault();  // присваиваем значение в контроллере  ViewData["NickName"] = personInfo.NickName;  return View();  }    View (razor syntax):   // используем значение в представлении  <h1>@ViewData["SomeProperty"]</h1> |

Характеристики ViewBag

1. ViewBag — динамическое свойство, представляет собой обертку вокруг ViewData и также используется для передачи данных из контроллера в соответствующее представление;
2. жизненный цикл также ограничивается текущий запросом;
3. при редиректе значение также оборачивается в null;
4. нет необходимости в приведении типов для получения данных

|  |
| --- |
| // запись во ViewBag    public ActionResult Index()  {   var personInfo = context.Personinfos.FirstOrDefault();   ViewBag.NickName = personInfo.NickName;     return View();  }    // используем значение в представлении  <h1>@ViewBag.NickName </h1> |

ViewBag имеет преимущество в том, что можно использовать его свойства Strongly typed-методом, при этом корректность данных будет проверена еще на этапе компиляции/ написания кода.

Тем не менее, по возможности следует избегать злоупотреблением ViewBag, ViewData и пользоваться стандартным для MVC способом — ViewModel.

Характеристики TempData

1. словарь данных, производный от TempDataDictionary, сохраняемый в коротких по времени жизни сессиях;
2. используется для передачи данных из текущего запроса в последующий запрос (для редиректа из одной страницы на другую);
3. жизненный цикл TempData длится до загрузки целевого представления;
4. необходимы приведение типов и проверка на null, чтобы избежать исключения;
5. используется только для хранения одноразовых сообщений, результатов исполнения операций и т.д.

|  |
| --- |
| // Основной запрос  public ActionResult EditAction(long recordId)  {   var personInfo = context.Personinfos.Single(p => p.id = recordId);   // сохраняем значение в TempData   TempData["personInfo"] = personInfo;   // помечаем значение для хранения до следующего запроса   TempData.Keep("personInfo");   // редиректим на другое действие   return RedirectToAction("EditRecord");  }    // сопутствующее действие  public ActionResult EditRecord()  {   PersonInfos personInfo;   if (TempData["personInfo"] != null)   {   // получаем и кастим объект   personInfo = TempData["personInfo"] as PersonInfos;   personInfo.NickName = "Pupkin Vasek";   }   // возвращаем данные в представление   return Json(personInfo);  } |

## ControllerContext и HttpContext.

Выполнение метода действия контроллера завершается созданием объекта, производного от класса ActionResult. В объекте ActionResult доступны два контекста MVC: ControllerContext и HttpContext. Отличие между ними состоит в том, что объект HttpContext описывает данные конкретного http-запроса, который обрабатывается приложеним, а ControllerContext описывает данные http-запроса по отношению к данному контролеру:

context.HttpContext.Response.Write("<div style='width:100%;text-align:center;'> <h2>" + context.Controller.ViewData["Capture"].ToString() + "</h2></div>");

В ASP.NET MVC как и в Web Forms получить данные о запросе: о браузере, об ip-адресе пользователя и т.д. можно через функциональность, инкапсулированную в объекте HttpContext.

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

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

Например:

|  |
| --- |
| 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:

|  |
| --- |
| 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 - свойство HttpContext.Response. Например, установим в куки значение "id":

|  |
| --- |
| 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 методы. 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:

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

Либо так:

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

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

***Работа с сеансом***

Чтобы сохранять и получать данные сеанса, нам надо воспользоваться свойством HttpContext.Session:

//запись в сессию

Session["NickName"] = personInfo.NickName;

// чтение из сессии.

string nickName = (string)(Session["NickName"]);

В данном случае, если у нас установлена на стороне клиента запись в сессии “NickName", то мы получим ее значение, используя соответствующее приведение типа. Перед чтением необходимо выполнять проверку на существование

***Определение пользователя***

Также объект HttpContext позволяет определить пользователя:

|  |
| --- |
| 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 |

## 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:

|  |
| --- |
| 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 асинхронный метод BookList возвращает не объект ActionResult, а объект Task<ActionResult>. Task представляет асинхронную операцию, выполняющуюся продолжительное время.

Чтобы обозначить метод как асинхронный, перед возвращаемым типом ставится ключевое слово async. В методе используется ключевое слово await, чтобы приостановить выполнение этого метода до тех пор, пока ожидаемая задача не завершится. В нашем случае такой задачей является получение данных из БД.

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 и модели

## 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:

|  |
| --- |
| @{      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:

|  |
| --- |
| #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:

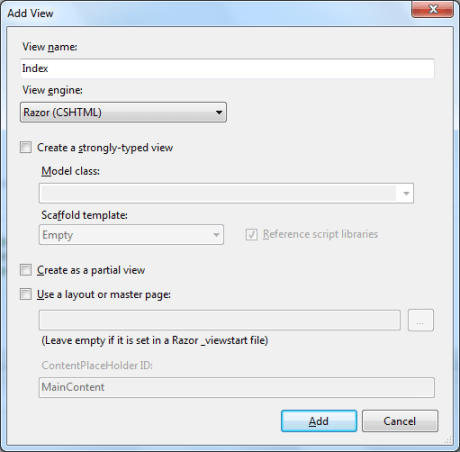
|  |
| --- |
| public ActionResult Index()  {      IEnumerable<Book> books = db.Books;      ViewBag.Books = books;      return View();  } |

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

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

***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:

|  |
| --- |
| @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:

|  |
| --- |
| @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 работоспособны, но есть и другой способ, более предпочтительный, который заключается в использовании строго типизированных представлений. Подобные представления позволяют передавать данные не через объект ViewBag, а напрямую в представление через параметр метода View. Код метода контроллера мог бы выглядеть так:

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

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

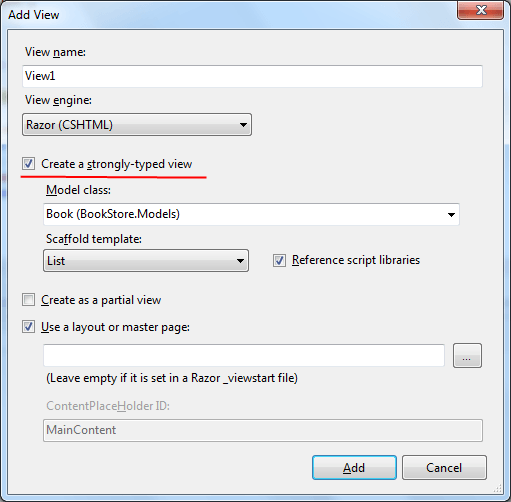
|  |
| --- |
| @model IEnumerable<BookStore.Models.Book>  @{      Layout = "~/Views/Shared/\_Layout.cshtml";  }      <div>          <h3>Распродажа книг</h3>          <table>              <tr class="header"><td><p>Название книги</p></td>                  <td><p>Автор</p></td>                  <td><p>Цена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

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

Чтобы не писать полностью имя типа модели, мы можем импортировать пространство имен в представлении:

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

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



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.

## 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.

Пример кода мастер-страницы \_Layout.chtml:

|  |
| --- |
| <!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:

|  |
| --- |
| @{      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:

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>      <meta charset="utf-8" />      <title>@ViewBag.Title</title>      <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css" />  </head>    <body>      <nav>          <ul class="menu">              <li>@Html.ActionLink("Главная", "Index", "Home")</li>          </ul>      </nav>      @RenderBody()      <footer>@RenderSection("Footer")</footer>  </body>  </html> |

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

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

Однако, при таком подходе, если у нас есть много 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:

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

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

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

*ViewStart*

Если у нас в проекте пара-тройка представлений, мы легко можем изменить 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:

|  |
| --- |
| @{      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

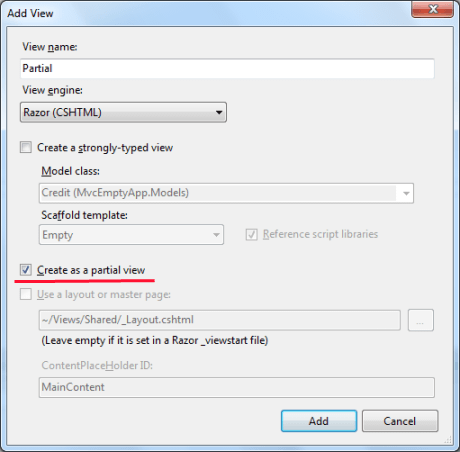
Кроме обычных представлений метод действия может также возвращать частичные представления. Их отличительной особенностью является то, что их можно встраивать в другие обычные представления. Частичные представления могут использоваться также как и обычные, однако наиболее удобной областью их использования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

Используются для включения подсекции разметки в несколько представлений. Частичные представления могут содержать код, вспомогательные методы HTML и ссылки на другие частичные представления. Частичные представления не могут вызывать методы действий, поэтому их нельзя использовать для выполнения бизнес-логики.

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

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

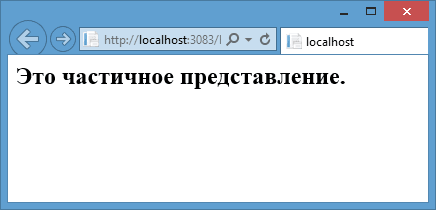
Теперь добавим новое представление Partial.cshtml. Для этого при создании представления в настройках укажем, что оно будет частичным:



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

<h2>@ViewBag.Message</h2>

После этого мы можем 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:

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

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

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

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

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

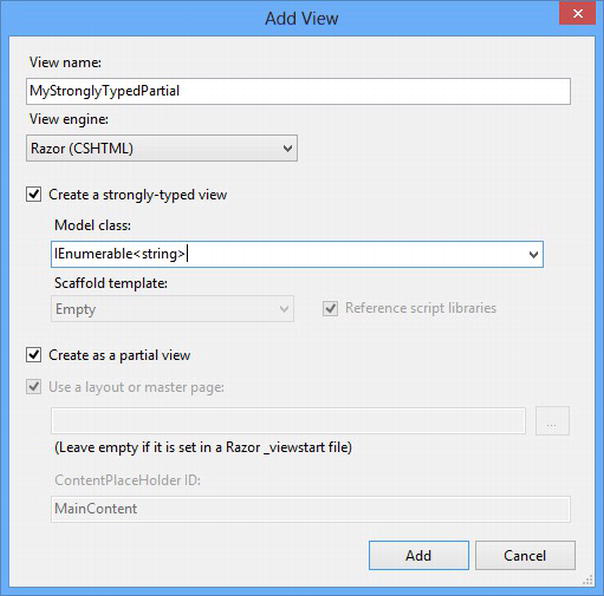
Также как и в случае с обычными представлениями, мы можем создавать строго типизированные частичные представления, указав в шапке файла директиву @model:

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

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

#### Строго типизированные частичные представления

Можно также создать строго типизированное частичное представление, а затем передавать в него объекты моделей представлений, которые оно будет визуализировать.



В листинге ниже показаны дополнения в файле частичного представления, который был создан Visual Studio. Изначально он содержит только тег модели @, который указывает тип модели представления.

Создаем строго типизированное частичное представление

@model IEnumerable<string>

**<div>**

**This is the message from the partial view.**

**<ul>**

**@foreach (string str in Model)**

**{**

**<li>@str</li>**

**}**

**</ul>**

**</div>**

Мы отображаем содержимое объекта модели представления как список HTML с помощью тега Razor @foreach.

Использование строго типизированного представления:

@{

ViewBag.Title = "List";

Layout = null;

}

<h3>This is the /Views/Common/List.cshtml View</h3>

@Html.Partial("MyStronglyTypedPartial", **new[] { "Apple", "Orange", "Pear" }**)

Здесь мы передаем дополнительный аргумент во вспомогательный метод Partial, который определяет объект модели представления.

## Работа с движком представления Razor

Главная цель представлений - визуализировать компоненты доменной модели как компоненты пользовательского интерфейса. Для этого нужно уметь добавлять в представления динамический контент. Динамический контент генерируется во время выполнения и может быть разным для разных запросов. Это его отличие от статического контента, такого как HTML, который вы создаете при написании приложения и который остается одинаковым для всех запросов. Динамический контент можно добавить в представления несколькими способами, которые описаны в таблице.

Способы добавления динамического контента в представления

| **Способ** | **Применение** |
| --- | --- |
| Код | Используется для создания небольших, независимых частей логики представления, например, операторы if или foreach. Это основной способ создания динамического контента в представлении, и на нем основаны некоторые другие подходы. |
| Секции | Используются для разбиения контента на блоки, которые будут вставлены в макет в определенных местах. |
| Дочерние действия | Используются для создания повторно используемых элементов управления UI и виджетов, в которых необходима бизнес-логика. Дочернее действие вызывает метод действия, визуализирует представление и внедряет результат в поток ответа |
| Вспомогательные методы HTML | Используются для создания одного элемента HTML или небольшой коллекции элементов, обычно на основании данных модели представления или объекта ViewData. Можно использовать встроенные вспомогательные методы MVC Framework, можно также создавать свои собственные. Вспомогательные методы HTML рассматриваются позже |

## Понятие движка представления

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:

|  |
| --- |
| <% 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, использующий декларативный синтаксис. Или Brail, который использует язык Boo.

***Код***

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

Например, переход к выражению кода:

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

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

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

|  |
| --- |
| <h3>@DateTime.Now.ToShortTimeString()</h3> |

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

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

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

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

Примеры первого способа приведены выше.

***Секции***

Движок Razor поддерживает концепцию секций, которые позволяют выделять различные области в макете. Секции Razor позволяют разбивать представление на части и контролировать то, где они внедряются в макет.

Определяем секцию в представлении:

@model string[]

@{

ViewBag.Title = "Index";

}

**@section Header {**

**<div class="view">**

**@foreach (string str in new[] { "Home", "List", "Edit" })**

**{**

**@Html.ActionLink(str, str, null, new { style = "margin: 5px" })**

**}**

**</div>**

**}**

<div class="view">

This is a list of fruit names:

@foreach (string name in Model)

{

<span><b>@name</b></span>

}

</div>

**@section Footer {**

**<div class="view">**

**This is the footer**

**</div>**

**}**

Секции определяются с помощью тега Razor @, за которым следует имя секции. В примере их две - Header и Footer. Их содержание представляет собой обычную смесь разметки HTML и тегов Razor.

Чтобы указать, где в макете должны отображаться секции, используется вспомогательный метод @RenderSection:

Используем секции в макете

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<meta name="viewport" content="width=device-width" />

<style type="text/css">

div.layout {

background-color: lightgray;

}

div.view {

border: thin solid black;

margin: 10px 0;

}

</style>

<title>@ViewBag.Title</title>

</head>

<body>

**@RenderSection("Header")**

<div class="layout">

This is part of the layout

</div>

**@RenderBody()**

<div class="layout">

This is part of the layout

</div>

**@RenderSection("Footer")**

<div class="layout">

This is part of the layout

</div>

</body>

</html>

Когда Razor анализирует макет, он заменяет вспомогательный метод RenderSection на содержимое секции представления с указанным именем. Те части представления, которые не содержатся в секциях, вставляются в макет с помощью вспомогательного метода RenderBody.

В представлении можно определить только те секции, на которые есть ссылки в макете. При попытке определить в представлении секции, для которых нет соответствующего вызова вспомогательного метода @RenderSection в макете, MVC Framework выбросит исключение.

Смешивание секций с остальным кодом представления – нестандартный подход. По соглашению секции определяются либо в начале, либо в конце представления, чтобы легче было увидеть, какие области контента будет рассматриваться как секции и будут захвачены вспомогательным методом RenderBody. Мы часто используем и другой подход, согласно которому представление должно содержать только секции, тело представления также заключается в секцию, как показано в следующем листинге.

Определяем представление с помощью секций Razor

@model string[]

@{

ViewBag.Title = "Index";

}

@section Header {

<div class="view">

@foreach (string str in new[] { "Home", "List", "Edit" })

{

@Html.ActionLink(str, str, null, new { style = "margin: 5px" })

}

</div>

}

**@section Body {**

**<div class="view">**

**This is a list of fruit names:**

**@foreach (string name in Model)**

**{**

**<span><b>@name</b></span>**

**}**

**</div>**

**}**

@section Footer {

<div class="view">

This is the footer

</div>

}

Такой подход часто позволяет создавать более понятные представления и снижает вероятность того, что RenderBody захватит посторонний контент. Чтобы его применить, мы должны заменить вызов вспомогательного метода RenderBody на RenderSection("Body"):

Определяем представление с помощью RenderSection("Body")

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<meta name="viewport" content="width=device-width" />

<style type="text/css">

div.layout {

background-color: lightgray;

}

div.view {

border: thin solid black;

margin: 10px 0;

}

</style>

<title>@ViewBag.Title</title>

</head>

<body>

@RenderSection("Header")

<div class="layout">

This is part of the layout

</div>

@RenderSection("Body")

<div class="layout">

This is part of the layout

</div>

@RenderSection("Footer")

<div class="layout">

This is part of the layout

</div>

</body>

</html>

*Тестирование секций*

Вы можете проверить, определена ли в представлении какая-либо секция из макета. Это делается для того, чтобы предоставить контент по умолчанию для этой секции, если она отсутствует в представлении и мы по каким-то причинам не хотим ее там определять:

Проверяем наличие секции Footer в представлении

@if (**IsSectionDefined("Footer")**)

{

@RenderSection("Footer")

}

else

{

<h4>This is the default footer</h4>

}

Вспомогательный метод IsSectionDefined возвращает true для заданного имени секции, если в текущем представлении определена эта секция. В этом примере с помощью вспомогательного метода мы выясняем, нужно ли предоставлять какой-либо контент по умолчанию, если окажется, что в представлении не определена секция Footer.

По умолчанию в представлении должны быть определены все секции, к которым в макете имеются вызовы RenderSection. Если секции отсутствуют, MVC Framework покажет пользователю исключение.

Добавляем в макет вызов RenderSection, для которого нет соответствующей секции в представлении

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<meta name="viewport" content="width=device-width" />

<style type="text/css">

div.layout {

background-color: lightgray;

}

div.view {

border: thin solid black;

margin: 10px 0;

}

</style>

<title>@ViewBag.Title</title>

</head>

<body>

@RenderSection("Header")

<div class="layout">

This is part of the layout

</div>

@RenderSection("Body")

<div class="layout">

This is part of the layout

</div>

@if (IsSectionDefined("Footer"))

{

@RenderSection("Footer")

}

else

{

<h4>This is the default footer</h4>

}

**@RenderSection("scripts")**

<div class="layout">

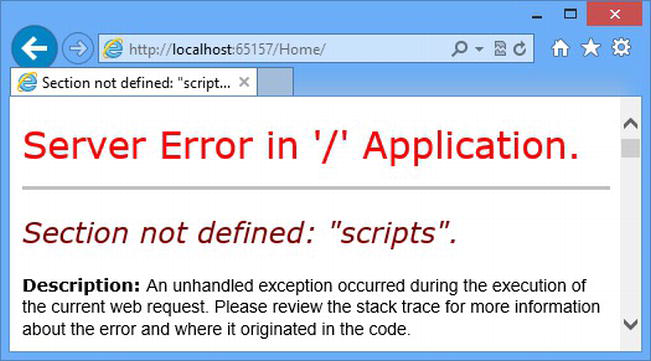
This is part of the layout

</div>

</body>

</html>

Если мы запустим приложение и движок Razor попытается визуализировать макет и представление, то увидим ошибку, показанную на рисунке.



Ошибка, которая появляется в случае отсутствия секции

Можно использовать метод IsSectionDefined, чтобы избежать вызовов RenderSection к секциям, не определенным в представлении, но для этого есть более элегантный подход: передавать дополнительное значение false в метод RenderSection:

Делаем секцию необязательной

@RenderSection("scripts", false)

Это делает секцию необязательной: если она определена в представлении, ее содержание будет вставлено в результат, если нет – мы не получим исключение.

***Дочерние действия***

Дочерние действия – это методы действий, которые вызываются из представления. Они позволяют избежать дублирования логики контроллера, которую необходимо использовать в приложении несколько раз. Дочерние действия так же относятся к действиям, как частичные представления – к представлениям.

Дочерние действия чаще всего используются для отображения какого-либо управляемого данными виджета, который должен появляться на нескольких страницах и содержит данные, не относящиеся к основному действию.

Любое действие можно использовать как дочернее.

Добавляем дочернее действие в контроллер Home

using System.Web.Mvc;

**using System;**

namespace WorkingWithRazor.Controllers

{

public class HomeController : Controller

{

public ActionResult Index()

{

string[] names = {"Apple", "Orange", "Pear"};

return View(names);

}

public ActionResult List()

{

return View();

}

**[ChildActionOnly]**

**public ActionResult Time()**

**{**

**return PartialView(DateTime.Now);**

**}**

}

}

Метод действия называется Time и визуализирует частичное представление, вызывая метод PartialView. Атрибут ChildActionOnly гарантирует, что метод действия может быть вызван только как дочернее действие из представления. Хотя метод действия может использоваться как дочерний и без этого атрибута, мы всегда применяем ChildActionOnly, чтобы данный метод невозможно было вызвать на запрос пользователя.

После определения метода действия нам нужно создать частичное представление, которое он будет визуализировать. Дочерние действия обычно связываются с частичными представлениями, хотя это и необязательно. В листинге ниже показано представление /Views/Home/Time.cshtml, которое мы создали для работы с нашим дочерним действием. Это строго типизированное частичное представление, моделью которого является объект DateTime.

Частичное представление для дочернего действия

@model DateTime

**<p>The time is: @Model.ToShortTimeString()</p>**

Дочернее действие вызывается вспомогательным методом Html.Action. После данного вызова выполняется метод действия, обрабатывается ViewResult, и вывод внедряется в ответ клиенту. В листинге ниже показаны изменения, которые мы внесли в файл /Views/Common/List.cshtml, чтобы визуализировать дочернее действие.

Вызов дочернего действия из представления List

@{

ViewBag.Title = "List";

Layout = null;

}

<h3>This is the /Views/Common/List.cshtml View</h3>

@Html.Partial("MyStronglyTypedPartial", new[] { "Apple", "Orange", "Pear" })

**@Html.Action("Time")**

Вспомогательному методу Action передан параметр с именем вызываемого метода действия. MVC Framework будет искать метод действия в контроллере, который обрабатывает текущий запрос. Чтобы вызвать метод действия из другого контроллера, передайте в параметр его имя, например:

@Html.Action("Time", "MyController")

В методы действий можно передавать параметры с помощью анонимно типизированных объектов, свойства которых соответствуют именам параметров дочернего метода действия. Так, например, следующее дочернее действие:

[ChildActionOnly]

public ActionResult Time(DateTime time)

{

return PartialView(time);

}

можно вызвать из представления следующим образом:

@Html.Action("Time", new { time = DateTime.Now })

## HTML-хелперы

Представления используют разметку html для визуализации содержимого. Однако фреймворк ASP.NET MVC обладает также таким мощным инструментом как HTML-хелперы, позволяющие генерировать html-код.

Это визуализаторы тегов HTML, помогающие создавать разметку страниц представлений.

Хелперы, инкапсулирующие код C#, который используется в разметке представлений, называются строчными.

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

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

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

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

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

|  |
| --- |
| @helper CreateList(string[] all)  {      <ul>          @foreach (string s in all)          {              <li>@s</li>          }      </ul>  }  @{      string[] cities = new string[] { "Лондон", "Париж", "Москва" };  }  @{      string[] countries = new string[] { "Великобритания", "Франция", "Россия" };  }  <h3>Города</h3>  @CreateList(cities)  <br />  <h3>Страны</h3>  @CreateList(countries) |

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:

|  |
| --- |
| 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:

|  |
| --- |
| @{      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 вполне можно использовать стандартные элементы html, например:

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

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

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

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

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

|  |
| --- |
| [HttpGet]  public ActionResult Buy()  {      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:

|  |
| --- |
| @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.

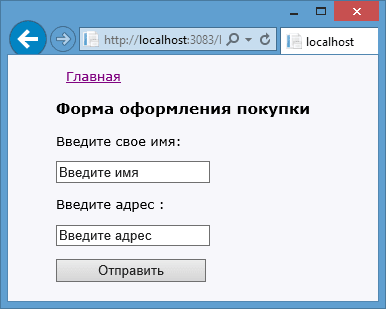
Вне зависимости от типа все базовые html-хелперы используют как минимум два параметра: первый параметр применяется для установки значений для атрибутов id и name, а второй параметр - для установки значения атрибута value

***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:

|  |
| --- |
| @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:

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

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

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

Этот хелпер сгенерирует следующую разметку:

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

***Html.Hidden***

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

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

сгенерирует разметку:

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

А при 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:

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

***Html.RadioButton***

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

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

Этот код создает следующую разметку:

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



***Html.CheckBox***

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

@Html.CheckBox("Enable", false)

Это выражение будет генерировать следующий HTML:

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

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

***Html.Label***

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

|  |
| --- |
| <label for="Name">Name</label> |

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

***Html.DropDownList***

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

Так, код

@Html.DropDownList("countires", new SelectList(new string[] {"Russia","USA", "Canada","France"}),"Countries")

генерирует следующую разметку:

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

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

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

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

Тогда в представлении мы можем так использовать этот SelectList:

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

И при рендеринге представления все элементы SelectList добавятся в выпадающий список

***Html.ListBox***

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

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

Этот код генерирует следующую разметку:

|  |
| --- |
| <select Length="9" id="countries" multiple="multiple" name="countires">  <option>Россия</option>  <option>США</option>  <option>Китай</option>  <option>Индия</option>  </select> |

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

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

Тогда метод контроллера мог бы получать эти значения следующим образом:

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

## 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. Например, в папке Models существует модель Purchase для оформления покупки книги:

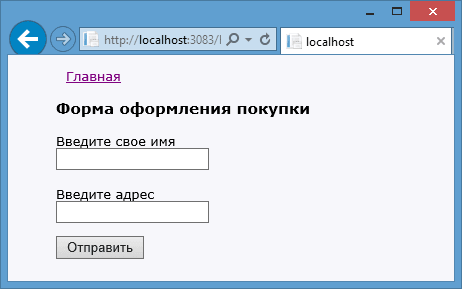
|  |
| --- |
| public class Purchase  {      public int PurchaseId { get; set; }      public string Person { get; set; }      public string Address { get; set; }      public int BookId { get; set; }      public DateTime Date { get; set; }  } |

И для ее использования может быть применена следующая форма:

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

Перепишем этот пример с использованием хелперов:

|  |
| --- |
| @model BookStore.Models.Purchase  @{      Layout = "~/Views/Shared/\_Layout.cshtml";  }        <div>          <h3>Форма оформления покупки</h3>          @using(Html.BeginForm("Buy", "Home", FormMethod.Post))          {              @Html.HiddenFor(m=>m.BookId)              @Html.LabelFor(m => m.Person,"Введите свое имя")              <br />              @Html.TextBoxFor(m=>m.Person)              <br /><br />              @Html.LabelFor(m => m.Address, "Введите адрес")              <br />              @Html.TextBoxFor(m=>m.Address)              <p><input type="submit" value="Отправить" /></p>          }      </div> |



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

Для каждого базового встроенного хелпера имеется свой строго типизированный хелпер:

* Html.CheckBoxFor

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

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

* Html.HiddenFor

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

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

* Html.LabelFor

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

<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>

## Шаблонные хелперы

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

В ASP.NET MVC имеются следующие шаблонные хелперы:

* Display

Создает элемент разметки для отображения значения указанного свойства модели: Html.Display("Name")

* DisplayFor

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

* Editor

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

* EditorFor

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

* DisplayText

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

* DisplayTextFor

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

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

* DisplayForModel

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

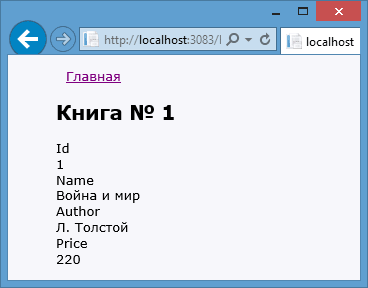
* EditorForModel

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

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

|  |
| --- |
| @{      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

Ранее было рассмотрено, как с помощью шаблонных хелперов отображать значения модели. Но логика приложения требует не только отображения модели, но и редактирования. Добавим в контроллер действие, которые будет получать по Id модель и выводить в представление ее свойства для редактирования:

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

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

Представление будет содержать набор хелперов EditorFor для некоторых полей модели:

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

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

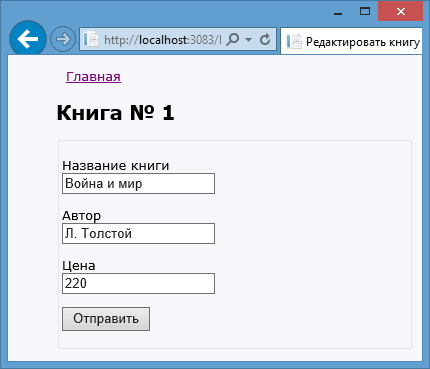
Теперь нужен сам код сохранения. Определим в контроллере действие EditBook, которое будет обрабатывать POST-запросы:

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

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

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

Обратимся к методу EditBook, например, Home/EditBook/1. Хелпер Html.EditorFor сгенерировал нам поля для редактирования. Мы можем изменить модель, и отправить ее на сервер, где произойдет ее сохранение:



## Добавление модели

Введение данных приложении часто выполняется через модели. Рассмотрим на примере как выполняется добавление модели. Для добавления модели вначале определим пару действий:

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

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

|  |
| --- |
| @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 можно переписать следующим образом:

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

## Удаление модели

Очень важно правильно организовать удаление модели, соблюдая правила безопасности. Рассмотрим потенциальные ошибки и проблемы приложения на простом примере. Добавим простое действие, которое удаляет модель из базы данных:

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

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

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

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

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

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

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

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

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

|  |
| --- |
| @{      Layout = "~/Views/Shared/\_Layout.cshtml";  }  @model BookStore.Models.Book  <h2>Удаление книги</h2>  <dl>      <dt>Название</dt>      <dd>          @Html.DisplayFor(model => model.Name)      </dd>        <dt>Автор</dt>      <dd>          @Html.DisplayFor(model => model.Author)      </dd>        <dt>Цена</dt>      <dd>          @Html.DisplayFor(model => model.Price)      </dd>  </dl>    @using (Html.BeginForm())  {      <input type="submit" value="Удалить"  />  } |

**Модели**

## Модели со сложной структурой

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

|  |
| --- |
| public class Player  {      public int Id { get; set; }      public string Name { get; set; }      public int Age { get; set; }      public string Position { get; set; }        public int? TeamId { get; set; }      public Team Team { get; set; }  }    public class Team  {      public int Id { get; set; }      public string Name { get; set; }      public string Coach { get; set; }        public IEnumerable<Player> Players { get; set; }  } |

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

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

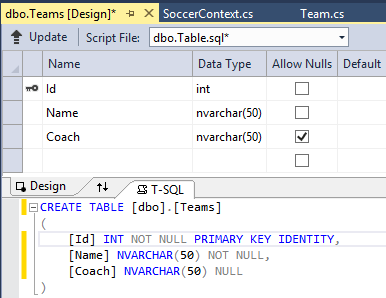
* Имя\_навигационного\_свойства+Имя ключа из связанной таблицы - в нашем случае имя навигационного свойства Team, а ключа из модели Team - Id, поэтому в нашем случае свойство называется TeamId.
* Имя\_класса\_связанной\_таблицы+Имя ключа из связанной таблицы - в нашем случае класс Team, а ключа из модели Team - Id. И здесь опять же получается TeamId.

Теперь создадим контекст данных, использующий модели:

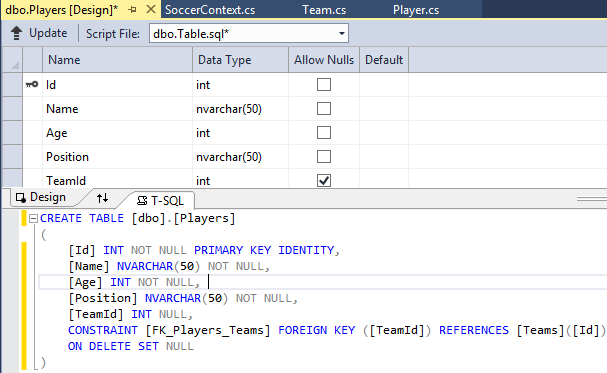
|  |
| --- |
| public class SoccerContext : DbContext  {      public DbSet<Player> Players { get; set; }      public DbSet<Team> Teams { get; set; }  } |

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

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



А таблица *Players* будет иметь следующую структуру:



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

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

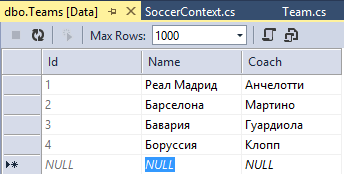
CONSTRAINT [FK\_Players\_Teams] FOREIGN KEY ([TeamId]) REFERENCES [Teams]([Id]) ON DELETE SET NULL

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

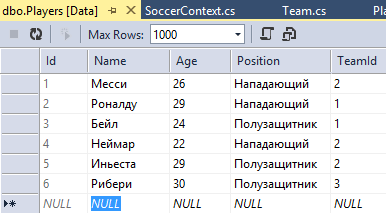
Последняя часть этого выражения (ON DELETE SET NULL) указывает, что при удалении объекта из таблицы Teams, свойству TeamId, которое ссылалось на удаленный объект, будет присвоено значение null.

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):



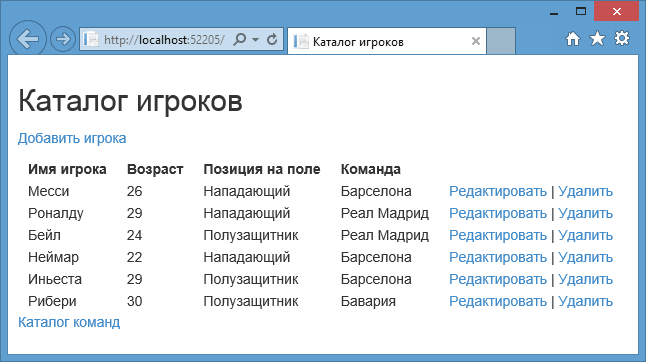
Теперь перейдем к созданию логики приложения. Добавим в приложение контроллер и определим в нем вывод всех игроков на страницу:

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Web;  using System.Web.Mvc;  using NavigationProperty.Models;  using System.Data.Entity;    namespace NavigationProperty.Controllers  {      public class 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 команду, связанную с данным игроком через внешний ключ. И создадим представление, которое будет выводить всех игроков:

|  |
| --- |
| @model IEnumerable<NavigationProperty.Models.Player>  @{      ViewBag.Title = "Каталог игроков";  }    <h2>Каталог игроков</h2>  <p>      @Html.ActionLink("Добавить игрока", "Create")  </p>  <table>      <tr>          <th>Имя игрока</th>          <th>Возраст</th>          <th>Позиция на поле</th>          <th>Команда</th>          <th></th>      </tr>        @foreach (var item in Model)      {          <tr>              <td>                  @Html.DisplayFor(modelItem => item.Name)              </td>              <td>                  @Html.DisplayFor(modelItem => item.Age)              </td>              <td>                  @Html.DisplayFor(modelItem => item.Position)              </td>              <td>                  @Html.DisplayFor(modelItem => item.Team.Name)              </td>              <td>                  @Html.ActionLink("Редактировать", "Edit", new { id = item.Id }) |                  @Html.ActionLink("Удалить", "Delete", new { id = item.Id })              </td>          </tr>      }  </table>  <p>      @Html.ActionLink("Каталог команд", "ListTeams")  </p> |

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



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

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

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

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

|  |
| --- |
| @using NavigationProperty.Models  @model Team    @{      ViewBag.Title = "Команда " + @Model.Name;      Layout = "~/Views/Shared/\_Layout.cshtml";  }    <div>      <h4>Команда @Model.Name</h4>      <hr />      <dl>          <dt>Название</dt>            <dd>              @Html.DisplayFor(model => model.Name)          </dd>            <dt>Тренер</dt>            <dd>              @Html.DisplayFor(model => model.Coach)          </dd>            <dt>Игроки</dt>            <dd>              <ul>                  @foreach (Player player in Model.Players)                  {                      <li>@player.Name (@player.Position)</li>                  }              </ul>          </dd>      </dl>  </div> |

## Передача массивов и сложных данных в контроллер

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

***Передача коллекции***

Определим следующую форму в представлении:

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

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

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

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

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

***Передача коллекции объектов модели***

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

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

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

|  |
| --- |
| @model List<BookStore.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)          @: Price: @Html.EditorFor(m => m[i].Price)      }      <input type="submit" />  } |

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

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

***Передача разных объектов одной модели***

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

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

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

|  |
| --- |
| @using BookStore.Models  @model Book  @{      ViewBag.Title = "Array";  }  @{      Book myBook = new Book() { Name = "Мартин Иден", Author = "Джек Лондон", Price = 190 };  }  <h2>Книги</h2>    @using (Html.BeginForm())  {      @Html.EditorFor(m=>myBook)      @Html.EditorForModel()      <input type="submit" />  } |

Одну модель мы передаем из контроллера в представление:

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

Вторую модель - myBook мы создаем уже в самом представлении. Все поля модели генерируются с помощью хелпера @Html.EditorFor(m=>myBook). Обратите внимание на имя модели - myBook. Так как мы ожидаем, что данный объект должен быть передан в метод в качестве параметра myBook, то он должен иметь точно такое же имя, а не произвольное.

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

# 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 находятся в пространстве имен System.ComponentModel.DataAnnotations (хотя несколько атрибутов определено в других пространствах).

***Атрибут Display***

Итак, предположим у нас имеется некоторая модель Book:

|  |
| --- |
| public class Book  {      public int Id { get; set; }      public string Name { get; set; }      public string Author { get; set; }      public int Year { get; set; }  } |

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

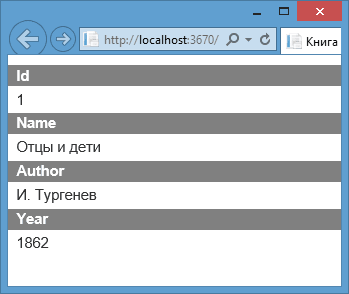
|  |
| --- |
| public class HomeController : Controller  {      BookContext db = new BookContext();      public ActionResult Index()      {          var firstBook = db.Books.FirstOrDefault();          return View(firstBook);      }  } |

Используя соответствующий хелпер, мы выводим эту модель представлении:

@Html.DisplayForModel():

|  |
| --- |
| @model DataAnnotations.Models.Book    @{      ViewBag.Title = "Книга";  }    @Html.DisplayForModel() |

Обычный код, но тут мы сталкиваемся с проблемой:

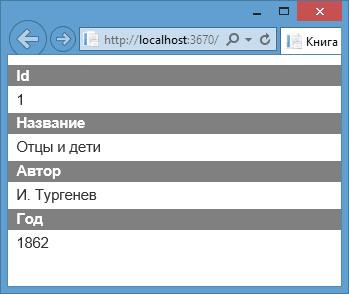


Как видно, заголовки у свойств точно такие же, как и названия свойств, что не очень удобно. Конечно, в этом случае мы могли бы использовать @Html.LabelFor для настройки названия для каждого отдельного свойства, но в этом случае у нас бы увеличился в несколько раз объем кода, и мы не смогли бы одной строкой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

Здесь окажутся очень полезными аннотации данных, а именно атрибут Display. Нам достаточно поправить модель, указав у нее параметры отображения:

|  |
| --- |
| using System.ComponentModel.DataAnnotations;  ...........................................      public class Book      {          public int Id { get; set; }          [Display(Name = "Название")]          public string Name { get; set; }          [Display(Name = "Автор")]          public string Author { get; set; }          [Display(Name = "Год")]          public 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:

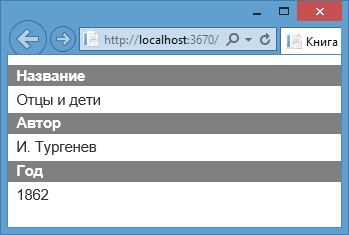


***Атрибут HiddenInput***

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:

|  |
| --- |
| using System.ComponentModel.DataAnnotations;  using System.Web.Mvc;  ...........................................      public class Book      {          [HiddenInput (DisplayValue=false)]          public int Id { get; set; }          [Display(Name = "Название")]          public string Name { get; set; }          [Display(Name = "Автор")]          public string Author { get; set; }          [Display(Name = "Год")]          public 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:

|  |
| --- |
| [ScaffoldColumn(false)]  public 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:

|  |
| --- |
| [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, а вместо них будут выводиться точки.

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

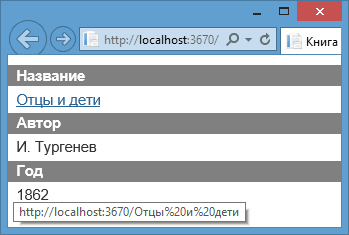
Хелперы редактирования создают текстовое поле

Хелперы отображения создают элемент ссылки для данного Url

Например, используем последний шаблон Url:

|  |
| --- |
| public class Book  {      [HiddenInput(DisplayValue = false)]      public int Id { get; set; }        [Display(Name = "Название")]      [UIHint("Url")]      public string Name { get; set; }        [Display(Name = "Автор")]      public string Author { get; set; }        [Display(Name = "Год")]      public int Year { get; set; }  } |

Тогда мы получим следующий результат:



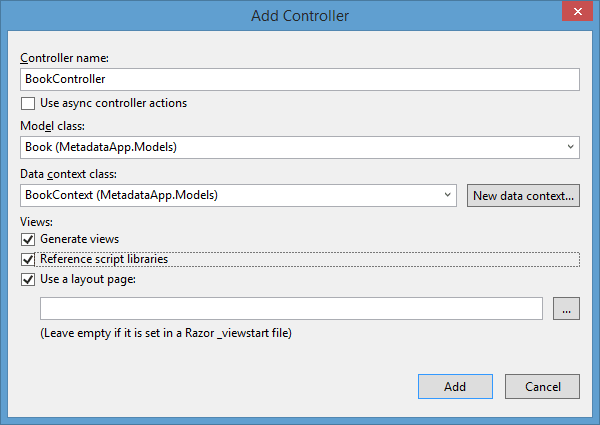
Таким образом, мы можем управлять отображением свойств модели в представлении.

## Основы валидации

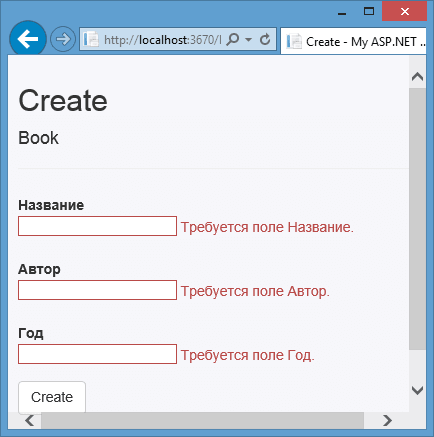
Для валидации модели Book добавим атрибуты к ее свойствам:

|  |
| --- |
| public class Book  {      [HiddenInput(DisplayValue = false)]      public int Id { get; set; }        [Required]      [Display(Name = "Название")]      public string Name { get; set; }        [Required]      [Display(Name = "Автор")]      public string Author { get; set; }        [Required]      [Display(Name = "Год")]      public int Year { get; set; }  } |

Далее создадим типизированный контроллер, который будет управлять объектами данной модели: в качестве шаблона выберем MVC 5 Controller with views, using Entity Framework, а в качестве класса модели укажем нашу модель Book, а в качестве контекста данных = BookContext:



В итоге будет по умолчанию создан набор представлений по управлению объектами модели. При вызове метода Create получим следующее представление:



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, благодаря чему и отображается сообщение об ошибке

|  |
| --- |
| <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. Чтобы настроить стиль отображения на стороне клиента, в файле Site.css по умолчанию указан стиль для соответствующих классов:

|  |
| --- |
| /\* styles for validation helpers \*/  .field-validation-error {      color: #b94a48;  }    .field-validation-valid {      display: none;  }    input.input-validation-error {      border: 1px solid #b94a48;  }    input[type="checkbox"].input-validation-error {      border: 0 none;  }    .validation-summary-errors {      color: #b94a48;  }    .validation-summary-valid {      display: none;  } |

1. Если в браузере отключен javascript, то сообщения об ошибках будут отсутствовать, а форма благополучно отправится на сервер. Чтобы не ограничиваться клиентской валидацией, во фреймворке предусмотрена также валидация на стороне сервера:

|  |
| --- |
| public ActionResult Create([Bind(Include="Id,Name,Author,Year")] 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.

Таков в общем механизм валидации приложения по умолчанию.

## Атрибуты валидации

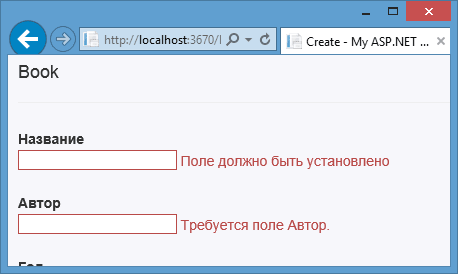
Рассмотрим атрибуты, участвующие в валидации модели.

***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:

|  |
| --- |
| public class Book  {      [HiddenInput(DisplayValue = 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; }        [Required]      [Display(Name = "Год")]      public int Year { get; set; }  } |



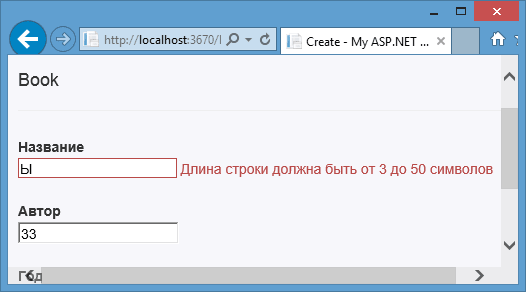
Таким образом, если мы явным образом не установим текст сообщения, то при выводе ошибки будет отображаться стандартный текст сообщения.

***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.

|  |
| --- |
| public class Book  {      [HiddenInput(DisplayValue = false)]      public 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 string Name { get; set; }        [Required]      [Display(Name = "Автор")]      public string Author { get; set; }        [Required]      [Display(Name = "Год")]      public int Year { get; set; }  } |



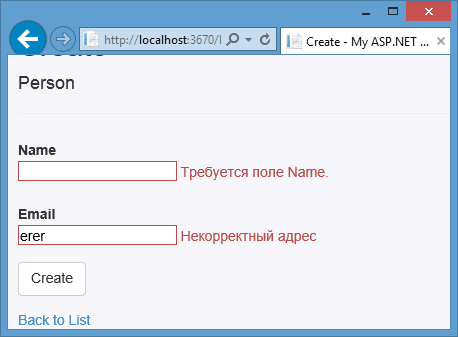
***Атрибут RegularExpression***

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:

|  |
| --- |
| [RegularExpression (@"[A-Za-z0-9.\_%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}", ErrorMessage = "Некорректный адрес")]  public 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:

|  |
| --- |
| [Display(Name = "Год")]  [Range(1700,2000,ErrorMessage="Недопустимый год")]  public 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).

|  |
| --- |
| [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 свойства UserName на сервер, а там оно уже сравнивается со значениями, находящимися в базе данных.

|  |
| --- |
| [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.

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

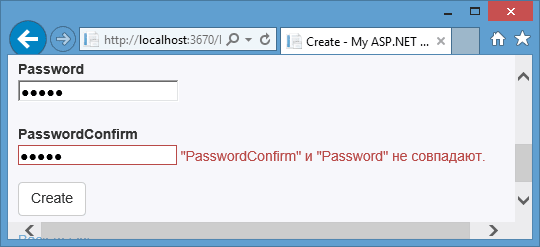
Это действие контроллера принимает в качестве параметра имя свойства, подлежащего валидации, и возвращает true или false в форме объекта в формате JSON.

***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:

|  |
| --- |
| [DataType(DataType.Password)]  public string Password { get; set; }    [Compare("Password",ErrorMessage="Пароли не совпадают")]  [DataType(DataType.Password)]  public  string PasswordConfirm { get; set; } |

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



Таким образом, используя атрибуты валидации модели, мы можем управлять валидацией и заключать несложную логику проверки значений свойств уже в атрибуты свойств, не прибегая к коду.

## Валидация 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:

|  |
| --- |
| [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:

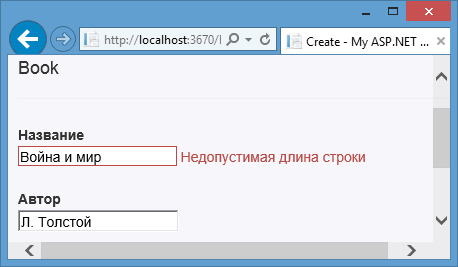
|  |
| --- |
| ModelState.IsValid == false  ModelState.IsValidField("Name") == false  ModelState["Name"].Errors.Count > 0 |

Мы также можем проверять корректность значений отдельных свойств модели:

|  |
| --- |
| [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:

|  |
| --- |
| <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,

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

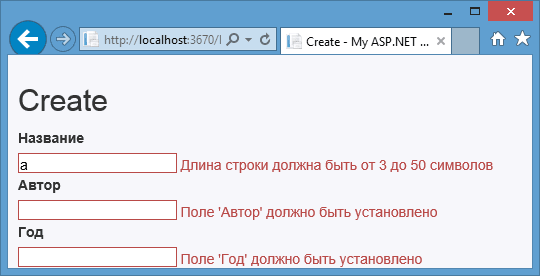
Оба свойства могут быть корректными в других ситуациях, но не вместе. Поэтому мы добавляем ошибку на уровне модели и в качестве первого параметра опускаем название свойства и указываем просто кавычки.

## Отображение ошибок валидации

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

|  |
| --- |
| @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:

|  |
| --- |
| @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***

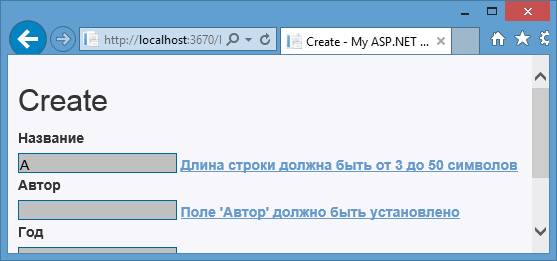
Еще один важный момент отображения ошибок - это их стилизация. То, что мы видим ошибки в красном цвете и границы полей ввода также в красном цвете, не жестко установлено, и мы все это можем изменить. В файле стилей Site.css мы можем найти секцию, которая как раз и отвечает за стилизацию:

|  |
| --- |
| /\* styles for validation helpers \*/  .field-validation-error {      color: #b94a48;  }   .field-validation-valid {      display: none;  }   input.input-validation-error {      border: 1px solid #b94a48;  }   input[type="checkbox"].input-validation-error {      border: 0 none;  }   .validation-summary-errors {      color: #b94a48;  }   .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:

|  |
| --- |
| .field-validation-error {      color: #6699CC;      font-weight:bold;      text-decoration:underline;  }    input.input-validation-error {      border: 1px solid #006699;      background-color: silver;  } |

В итоге при возникновении ошибок валидации мы увидим следующее представление:



## Создание собственной логики валидации

Хотя встроенные атрибуты валидации охватывают приличное количество ситуаций, которые могут возникнуть, но все же их бывает недостаточно. Но фреймворк mvc 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 имен System.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:

|  |
| --- |
| 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:

|  |
| --- |
| public class Book  {      [HiddenInput(DisplayValue = false)]      public int Id { get; set; }        [Required (ErrorMessage="Поле должно быть установлено")]      [Display(Name = "Название")]      public string Name { get; set; }        [Required]      [MyAuthors(new string[] { "Л. Толстой", "А. Пушкин", "Ф. Достоевский", "И. Тургенев" }, ErrorMessage="Недопустимый автор")]      [Display(Name = "Автор")]      public string Author { get; set; }        [Required]      [Display(Name = "Год")]      [Range(1700,2000,ErrorMessage="Недопустимый год")]      public int Year { get; set; }  } |

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

***Атрибуты валидации на уровне модели***

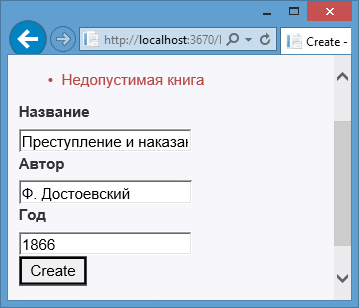
Атрибуты валидации на уровне модели применяются к проверке комбинации свойств. Например, мы не хотим, чтобы пользователь мог ввести определенную книгу:

|  |
| --- |
| 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:

|  |
| --- |
| [NotAllowedAttribute(ErrorMessage="Недопустимая книга")]  public class Book  {      [HiddenInput(DisplayValue = false)]      public int Id { get; set; }        [Required (ErrorMessage="Поле должно быть установлено")]      [Display(Name = "Название")]      public 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**:

|  |
| --- |
| public class Book : IValidatableObject  {      [HiddenInput(DisplayValue = false)]      public int Id { get; set; }        [Display(Name = "Название")]      public string Name { get; set; }        [Display(Name = "Автор")]      public string Author { get; set; }        [Display(Name = "Год")]      public int? Year { get; set; }        public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)      {          List<ValidationResult> errors = new List<ValidationResult>();            if (string.IsNullOrWhiteSpace(this.Name))          {              errors.Add(new ValidationResult("Введите название книги"));          }          if (string.IsNullOrWhiteSpace(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.

|  |
| --- |
| 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:

|  |
| --- |
| 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 объект ModelValidationResult 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 свойства MemberName мы передаем пустые кавычки:

|  |
| --- |
| 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:

|  |
| --- |
| public class MvcApplication : System.Web.HttpApplication  {      protected void Application\_Start()      {          ModelValidatorProviders.Providers.Add(new MyValidationProvider());            Database.SetInitializer(new BookDbInitializer());            AreaRegistration.RegisterAllAreas();          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:

|  |
| --- |
| public class Book  {      [HiddenInput(DisplayValue = false)]      public int Id { get; set; }        [Display(Name = "Название")]      public string Name { get; set; }        [Display(Name = "Автор")]      public string Author { get; set; }        [Display(Name = "Год")]      public int Year { get; set; }  } |

# Привязка модели

## Введение в привязку моделей

Ранее уже было рассмотрено 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:

|  |
| --- |
| [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:

|  |
| --- |
| [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:

|  |
| --- |
| 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**:

|  |
| --- |
| 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:

|  |
| --- |
| 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:

|  |
| --- |
| 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 Book имеет 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 класса System.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:

|  |
| --- |
| 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:

|  |
| --- |
| 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

|  |
| --- |
| [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; }  } |

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 UpdateModel выбрасывает исключение. Ниже показан пример использования метода UpdateModel в действии Edit вместо применения параметра:

|  |
| --- |
| [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.

|  |
| --- |
| [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");      }  } |

# 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 — процесс определения идентичности пользователя и обеспечения гарантий этой идентичности. Процесс аутентификации аналогичен регистрации участников конференции. Во-первых, вы предъявляете некоторое свидетельство, доказывающее вашу идентичность (вроде паспорта или водительских прав). Во-вторых, как только ваша идентичность проверена по этой информации, вы получаете личный значок участника конференции, или маркер, который постоянно носите с собой на протяжении всей конференции. Любой, кто вас встретит на конференции, сможет легко определить вашу идентичность, взглянув на этот значок, который обычно содержит базовую идентифицирующую информацию, такую как ваше имя и фамилия. Весь этот процесс — пример аутентификации. Как только ваша идентичность установлена, ваш маркер подтверждает ее, так что куда бы вы ни пошли в пределах конкретной области, ваша личность известна.

В приложениях ASP.NET аутентификация реализуется одной из четырех возможных аутентифицирующих систем:

• аутентификация Windows;

• аутентификация с помощью форм;

• аутентификация с помощью паспортов;

• специальный процесс аутентификации.

В каждом случае пользователь предъявляет некоторое удостоверение при регистрации в системе. Идентичность пользователя отслеживается разными способами, в зависимости от типа аутентификации. Например, операционная система Windows использует 96-битное число, называемое SID (security identifier — идентификатор безопасности) для идентификации каждого входящего пользователя. В аутентификации форм ASP.NET пользователю выдается аутентифицирующий мандат формы, представляющий собой комбинацию значений, которые шифруются и помещаются в cookie-набор.

Вся аутентификация позволяет приложению идентифицировать, какой пользователь присылает каждый запрос. Это хорошо работает для персонализации и пользовательской настройки, потому что вы можете использовать идентифицирующую информацию для выдачи специфичных для пользователя сообщений на Web-странице, изменять внешний вид Web-сайта, добавлять специальное содержимое на базе предпочтений конкретного пользователя и тому подобное. Однако аутентификации самой по себе недостаточно для ограничения задач, которые разрешено выполнять пользователю на базе его идентичности. Для этого нужна авторизация, о которой мы поговорим ниже.

Авторизация — процесс определения прав и ограничений, назначенных аутентифицированному пользователю. Если взять аналогию с конференцией, то авторизация — это процесс выдачи допуска на определенные мероприятия, например, на главный доклад. На большинстве конференций можно подписаться на разные уровни доступа — на полный доступ, только вступительное заседание или только на посещение выставочного зала. Это значит, что если вы захотите посетить главное заседание “Конференции профессиональных разработчиков Microsoft”, чтобы послушать, что скажет Билл Гейтс, то должны для этого иметь соответствующие права доступа. Как только вы войдете в зал заседаний, сотрудник службы безопасности проверит вашу эмблему участника конференции. На основании указанной на ней информации он либо пропустит вас, либо скажет, что вы не можете войти. Это пример авторизации. В зависимости от идентифицирующей вас информации, вам либо открывается, либо закрывается доступ к запрашиваемым ресурсам.

Пример с конференцией представляет случай авторизации на базе ролей — когда авторизация определяется правами группы, к которой принадлежит пользователь, а не на том, кто он такой. Другими словами, вы авторизуетесь для доступа в зал заседаний на основании роли (типа допуска), а не вашей специфичной идентифицирующей информации (имени и фамилии). Во многих случая авторизация на базе ролей предпочтительна, поскольку ее гораздо легче реализовать. Если сотрудник безопасности должен будет сверять имя каждого гостя со списком допущенных, то процесс авторизации существенно замедлится. То же самое верно и для Web-приложений, хотя более вероятно, что роли будут следующими: менеджеры, администраторы, гости, продавцы, клиенты и т.д.

Существует несколько причин применения фиксированной учетной записи для запуска кода ASP.NET. Почти во всех приложениях права, выданные пользователю, не соответствуют правам, которые требуются приложению, работающему от имени пользователя. Как правило, ваш код нуждается в более широком наборе привилегий, чтобы выполнять задачу идентификации, и вы не захотите выдавать такие права каждому пользователю, который может обращаться к вашему приложению. Например, вашему коду может понадобиться создавать журнальные записи о возможных сбоях, даже если данному пользователю не разрешено напрямую писать в журнал событий Windows, в файл или в базу данных. Аналогично приложения ASP.NET всегда должны иметь право доступа в каталог c:\[WinDir]\Microsoft.NET\[Version]\Temporary ASP.NET Files, чтобы создавать и кэшировать версии ваших Web-страниц на машинном языке.

Для доступа к методу только авторизованого пользователя используется фильтр авторизации в виде атрибута Authorize, который устанавливается либо для одиночного действия контроллера, либо для всего контроллера.

|  |
| --- |
| [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.

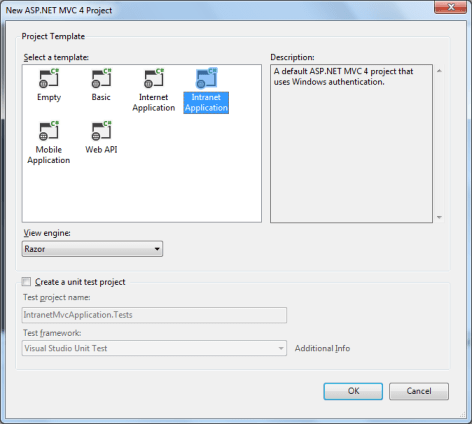
Профили представляют различную информацию о пользователях, как например, имя, фамилию, рост, возраст и т.д.

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:

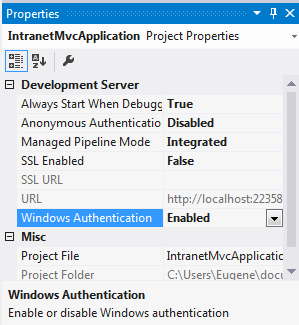


По сути Intranet-приложение похоже на шаблон Internet Application, поскольку имеет уже готовую инфраструктуру. Однако оно нацелено именно на использование аутентификации Windows, поэтому мы не увидим в отличие от Internet Application контроллера AccountContoller и моделей, представляющих учетные записи.

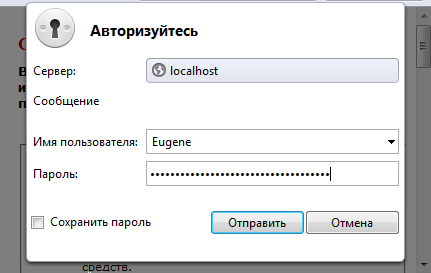
Аутентификация Windows задана следующей строкой в файле *web.config*:

|  |
| --- |
| <authentication mode="Windows" /> |

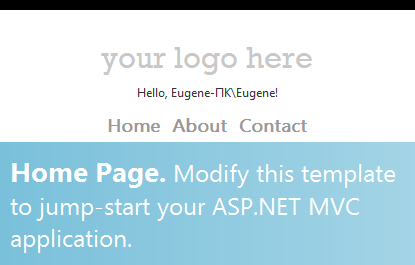
Теперь настроим приложение на данный способ аутентификации. Для этого перейдем к свойствам проекта и установим для свойства Anonymous Authentication значение Disabled, а для свойства Windows Authentication значение Enabled:



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:

|  |
| --- |
| [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*:

|  |
| --- |
| <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.

В Visual Studio 2012 этот метод будет выглядеть следующим образом:

|  |
| --- |
| [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

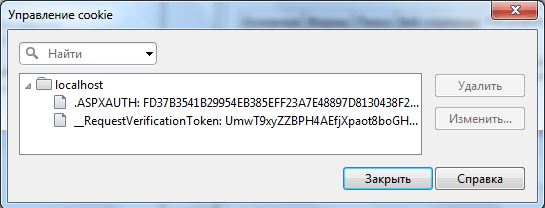
В этом и по сути и состоит сущность аутентификации форм.

Несмотря на то, что 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:

|  |
| --- |
| 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:



Видно что, 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:

|  |
| --- |
| <system.web>    ..................    <authentication mode="Forms">      <forms loginUrl="~/Account/Login" timeout="600" name="cookies" />    </authentication>    ............  </system.web> |

***Контекст безопасности***

Независимо от системы аутентификации, ASP.NET использует в основе одну и ту же модель представления информации о пользователе и ролях. Пользователям, которые регистрируются в Web-приложении, на основе предъявленных ими удостоверений выделяется принципал (principal) и идентичность. Объект-принципал представляет текущий контекст безопасности данного пользователя. Он комбинирует самого пользователя (идентичность) с информацией, сохраняемой в учетной записи текущего пользователя — такой как роли, привилегии и многое другое. Таким образом, это позволяет вам выполнять авторизацию, основанную на ролях, и предоставляет ссылку на соответствующий объект идентичности. Объект идентичности представляет успешно авторизованного пользователя и потому предоставляет информацию о пользователе наподобие его имени.

***Интерфейс IPrincipal***

Все объекты-принципалы реализуют интерфейс IPrincipal, который определяет центральный набор функциональности. Когда вы обращаетесь к свойству User текущей Web-страницы (System.Web.UI.Page), или из текущего контекста HTTP (HttpContext.Current), то получаете доступ к объекту IPrincipal, представляющему контекст безопасности текущего пользователя.

Интерфейс IPrincipal определяет единственное свойство по имени Identity, которое извлекает объект IIdentity, представляющий информацию о текущем пользователе. Интерфейс IPrincipal также определяет единственный метод по имени IsInRole(), позволяющий проверить, является ли текущий пользователь членом специфической роли.

Рассмотрим пример, использующий метод IsInRole() для проверки того, является ли текущий пользователь членом роли под названием Admin:

if (HttpContext.User.IsInRole("Admin"))

{

// todo

}

При использовании аутентификации Windows или аутентификации с помощью форм объект-принципал создается автоматически. Однако этот объект можно также создать “на лету”, с информацией о пользователе и роли, извлеченной из другого места, такого как пользовательская база данных.

***Интерфейс IIdentity***

Подобно интерфейсу IPrincipal, интерфейс IIdentity представляет соглашение, не зависящее от используемой схемы аутентификации. Все объекты идентичности должны реализовывать IIdentity.

Интерфейс IIdentity определяет базовую информацию, необходимую для представления текущего пользователя. Как минимум, это включает в себя три следующих свойства, доступных только для чтения.

• AuthenticationType. Возвращает тип используемой аутентификации как строку (forms, Passport, NTLM или пользовательский тип аутентификации).

• IsAuthenticated. Возвращает булевское значение, указывающее на то, был ли пользователь аутентифицирован (true), или же он является анонимным (false).

• Name. Возвращает имя текущего пользователя в виде строки.

Вы можете обращаться к объекту IIdentity, представляющему текущего пользователя, через объект IIPrincipal. Ниже показан пример, использующий эту технику для проверки того, был ли пользователь аутентифицирован:

if (HttpContext.User.Identity.IsAuthenticated)

{

// todo

}

## 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.

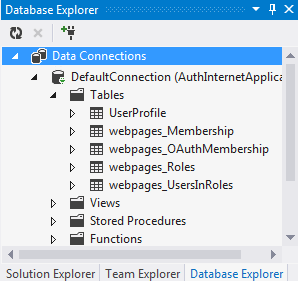
InitializeSimpleMembershipAttribute содержит определение класса-инициализатора SimpleMembershipProvider, который создает соединение с базой данных следующим вызовом:

|  |  |
| --- | --- |
| 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:

|  |
| --- |
| //..................................  internal static string MembershipTableName  {      get { return "webpages\_Membership"; }  }    internal static string OAuthMembershipTableName  {      get { return "webpages\_OAuthMembership"; }  }  //........................ |

Другие две таблицы закодированы в коде SimpleRoleProvider:

|  |
| --- |
| internal static string RoleTableName  {      get { return "webpages\_Roles"; }  }    internal static string UsersInRoleTableName  {      get { return "webpages\_UsersInRoles"; }  } |

Чтобы сменить способ организации данных в таблицах, придется создать собственные провайдеры членства и ролей. (Просмотреть код провайдеров можно по следующим ссылкам: [SimpleRoleProvider](http://aspnetwebstack.codeplex.com/SourceControl/changeset/view/e0115a823029" \l "src/WebMatrix.WebData/SimpleRoleProvider.cs) и [SimpleMembershipProvider](http://aspnetwebstack.codeplex.com/SourceControl/changeset/view/e0115a823029" \l "src/WebMatrix.WebData/SimpleMembershipProvider.cs)

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.

## 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:

|  |
| --- |
| <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:

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

Не забываем и про настройку аутентификации в глобальном web.config:

<system.web>

  ..................

  <authentication mode="Forms">

    <forms loginUrl="~/Account/Login" timeout="600" name="cookies" />

  </authentication>

  ............

</system.web>

Теперь создадим простейшую инфраструктуру для нашего проекта. В начале добавим модели логина и регистрации в папку Models:

|  |
| --- |
| 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:

|  |
| --- |
| 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.

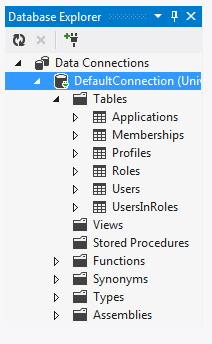
Теперь создадим представления, которые будут регистрировать пользователей и осуществлять вход. Итак, представление регистрации *Register.cshtml*:

|  |
| --- |
| @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>  } |

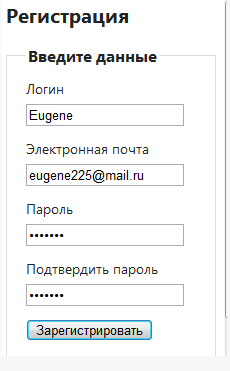
И представление логина *Login.cshtml*:

|  |
| --- |
| @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:



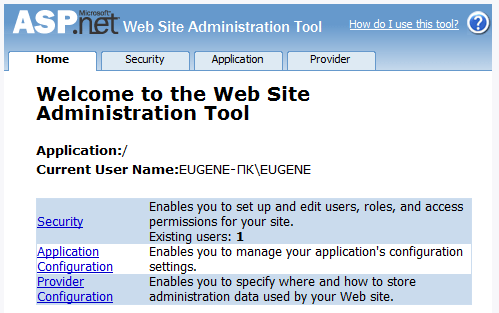
Зарегистрируем нового пользователя:



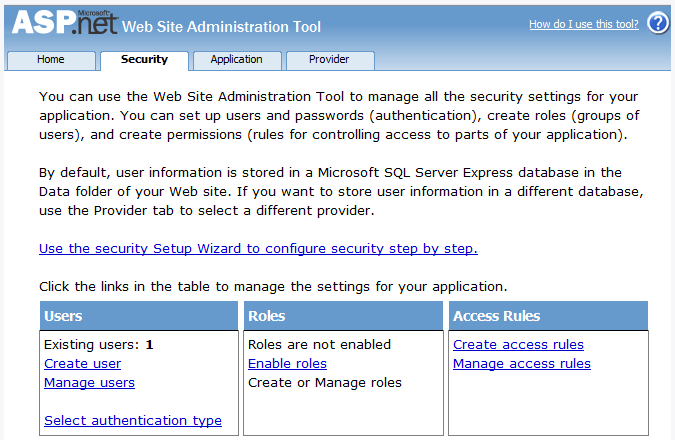
После этого введенная информация попадет в базу данных.

## Web Site Administration Tool

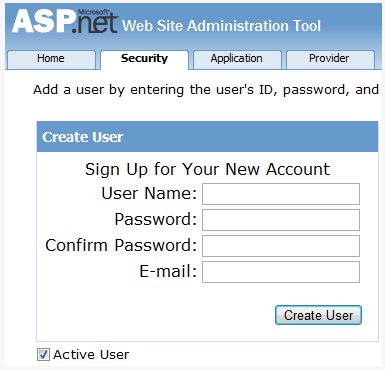
Кроме добавления и изменения настроек учетных записей через формы регистрации, изменения пароля и другие, мы можем использовать средство администрирования Web Site Administration Tool - графический интерфейс для управления членство и ролями. Для этого выберем в меню Project пункт ASP.NET Configuration. После этого у нас откроется в веб-браузере страница администрирования веб-приложения:



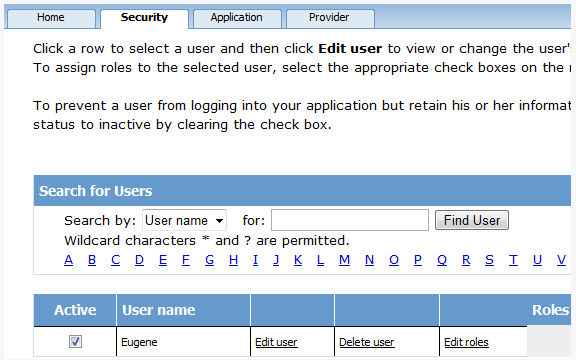
На вкладке Security мы можем произвести управление учетными записями: создать, изменить или удалить пользователей, добавить роли и установить для пользователей роли:



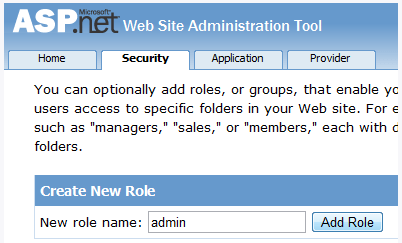
Для создания нового пользователя выберем ссылку Create user. После этого нам откроется форма создания нового пользователя:



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.