5) Основные концепции ASP.NET Web Forms // Способы проверки данных в ASP.NET MVC

Основные концепции ASP.NET Web Forms

Вначале рассмотрим схему работы в сети Интернет, которую можно назвать классической, так как эта схема является исторически первой. Основными элементами классической схемы являются *браузер* и *веб-сервер*. При взаимодействии браузера и веб-сервера выполняются следующие шаги:

1. Браузер формирует *запрос* к серверу, используя *протокол HTTP*. Как правило, браузер запрашивает *HTML-страницу*, то есть текстовый файл, содержащий HTML-код.
2. Сервер анализирует запрос браузера и извлекает из локального хранилища требуемый файл.
3. Сервер формирует *HTTP-ответ*, включающий требуемую информацию, и отсылает его браузеру по протоколу HTTP.
4. Браузер выполняет отображение страницы.

Классическая схема проста, но обладает существенным недостатком – страницы статичны, и их содержимое не может меняться на сервере в зависимости от запросов клиента. Все большее распространение получают технологии, при использовании которых страницы (целиком или частично) генерируются на сервере *непосредственно* перед отправкой клиенту. Работают технологии «серверных страниц» по схожим принципам:

- Для представления информации на сайте используются не страницы с HTML-кодом, а серверные страницы специального синтаксиса (который часто является HTML-подобным).
- При запросе серверной страницы веб-сервер запускает отдельный служебный процесс, которому перенаправляется запрос.
- В служебном процессе страница анализируется, по ней генерируется некий объект, соответствующий странице.
- Служебный процесс выполняет методы сгенерированного объекта. Как правило, объект имеет специальный метод, генерирующий выходной поток страницы в виде HTML-кода.
- Выходной HTML-поток перенаправляется веб-серверу, который формирует HTTP-ответ и отсылает его браузеру.
- Браузер выполняет отображение страницы.

Далее выделим особенности, присущие технологии ASP.NET.

- *Работа с управляемым кодом.* Служебный процесс ASP.NET основан на управляемом коде. Запросы к каждому веб-приложению обрабатываются в отдельном домене служебного процесса. Серверной странице ставиться в соответствие пользовательский класс, объект которого непосредственно генерирует страницу. Также доступны стандартные для .NET библиотеки классов и возможности межъязыкового взаимодействия.
- *Разделение кода и представления*. Данная концепция также называется *Code Behind*. Согласно ей желательно, чтобы страница ASP.NET состояла из двух частей: файла с описанием вида страницы (разметка, элементы управления) и файла с кодом методов

страницы. Эти два файла могут компилироваться в отдельные классы или представлять собой частичный класс. При изменении любого из файлов на сервере происходит перекомпиляция страницы.

- *Серверные элементы управления.* Для конструирования страницы ASP.NET содержит несколько десятков специальных серверных элементов управления. Каждый такой элемент в конечном итоге транслируется в один или несколько обычных элементов HTML. Серверные элементы управления поддерживают событийную модель, содержат большое количество настраиваемых свойств. Они предоставляют более высокий уровень абстракции в сравнение с классическими элементами управления HTML. Кроме этого, имеется возможность создавать собственные серверные элементы управления.
- *Событийная модель*. Технология ASP.NET пытается перенести на веб-программирование принципы, используемые при написании приложений Windows Forms. Речь идет о программировании, основанном на обработке событий. Отдельный серверный элемент управления ASP.NET, как правило, обладает набором некоторых событий. Например, у элемента `Button` (кнопка) есть событие `OnClick`. Для того чтобы закодировать логику страницы, программист пишет обработчики соответствующих событий. Когда событие происходит, информация о нём пересылается от клиента на сервер, где срабатывает обработчик события. Затем страница вновь пересылается клиенту. Подчеркнем следующие важные детали. Во-первых, основой реализации событийной модели является схема, при которой страница отсылает запросы сама к себе. Во-вторых, чтобы страница сохраняла свое состояние между отдельными циклами приёма-передачи, это состояние фиксируется в специальном скрытом поле страницы. Этот технологический прием называется в ASP.NET *поддержкой состояния представления страницы* (коротко – *поддержка View State*). И, наконец, ASP.NET пытается перенести событийную модель на возможно большее количество классических элементов управления HTML. Для реализации этого используются «вкрапления» в страницу клиентских скриптов.
- *Поддержка пользовательских сессий и кэширование.* В ASP.NET существует богатый набор встроенных возможностей для работы с данными пользовательских сессий, выполнения кэширования данных, идентификации пользователей.

С выходом платформы .NET версии 2.0 технология ASP.NET получила значительное число изменений и улучшений по сравнению с первой версией. Вот основные из них.

- *Поддержка динамической компиляции*. ASP.NET 2.0 предоставляет возможность размещать *исходный код* классов в специальном подкаталоге веб-приложения `App_Code`, а при обращении к классам динамически компилирует их.
- *Эталонные страницы и темы.* В ASP.NET 2.0 задача создания прототипов страниц решена путём введения концепции *эталонной страницы* (*master page*). Разработчики сайтов с большим количеством страниц, имеющих единообразную схему и однотипную функциональность, могут теперь программировать все это в одном эталонном файле, вместо того чтобы добавлять информацию об общей структуре в каждую страницу. *Тема ASP.NET* представляет собой комплекс настраиваемых стилей и визуальных атрибутов элементов сайта. Тема идентифицируется именем и состоит из CSS-файлов, изображений и обложек элементов управления. *Обложка* (*skin*) элемента управления - это текстовый файл, который содержит используемое по умолчанию объявление данного элемента со значениями его свойств.

- *Адаптивный рендеринг.* В ASP.NET 2.0 реализована адаптерная архитектура элементов управления, позволяющая одному и тому же элементу по-разному осуществлять свой рендеринг в зависимости от типа целевого браузера. Адаптерная архитектура позволяет создавать собственные адаптеры, настраивая серверные элементы управления для использования с определенными браузерами.
- *Модель поставщиков.* В основу модели поставщиков ASP.NET 2.0 положена известная архитектурная концепция – шаблон проектирования «стратегия». Особенностью шаблона является то, что он даёт объекту или целой подсистеме возможность открыть свою внутреннюю организацию таким образом, чтобы клиент мог отключить используемую по умолчанию реализацию той или иной функции и подключить другую её реализацию, в том числе собственную. Реализация модели поставщиков в самой ASP.NET дает возможность настраивать определенные компоненты её исполняющей среды. Для этой цели здесь определены специальные классы поставщиков, которые можно использовать в качестве базовых классов при создании собственных поставщиков.
- *Новые элементы управления*. В дополнении к существующим, ASP.NET 2.0 предлагает набор новых элементов управления, в частности, для представления деревьев, навигации по сайту, работы с данными пользователя.

Любая страница *ASP.NET* представлена классом, производным от класса `System.Web.UI`, который определяет свойства, методы и события, общие для всех страниц, предназначенных для обработки средой *ASP.NET*. Наиболее важные свойства этого объекта приведены в таблице ниже:

Свойство	Описание
`Application`	Возвращает объект `HttpApplicationState`
`Cache`	Возвращает объект `Cache`, в котором хранятся данные приложения, в т.ч. и самой страницы
`IsPostBack`	Возвращает значение, определяющее, была ли страница загружена клиентом впервый раз, или загружена повторно в ответ на запрос клиента
`Request`	Возвращает объект `HttpRequest`, используемый для получения информации о входящем запросе HTTP
`Response`	Возвращает объект `HttpResponse`, используемые для формирования ответа сервера клиенту
`Server`	Возвращает объект `HttpServerUtility`
`Session`	Возвращает объект `System.Web.SessionState.HttpSessionState`, с помощью которого получается информация о текущем сеансе HTTP

Такое построение проекта позволяет хранить отдельно *код представления* для генерации *HTML* кода (в файле `*.aspx`) от *программной*

логики (в файле `*.aspx.cs`), что во многих случаях существенно упрощает разработку сложных веб-приложений.

ASP.NET Web Forms – это механизм, который работает на базе общей среды исполнения *ASP.NET* и встроен в поставку *ASP.NET* начиная с самой первой версии. Механизм веб-форм предполагает построение веб-приложений аналогично тому, как мы привыкли их строить в случае с настольными приложениями. Этот механизм пытается исключить различия между веб-приложениями и настольными приложениями, сделав процесс их построения максимально похожим.

Результат работы веб-приложения – это код *HTML*, который передается клиенту. Обычно *HTML*-код страницы веб-приложения содержит различные элементы, которые позволяют управлять процессом работы приложения – например, кнопки, поля ввода, переключатели и т.д. Эти *элементы управления* можно описать на языке *HTML*.

Элемент управления *ASP.NET Web* Forms – это *объект* .*NET* Framework, который является наследником базового класса `Control` и который реализует в себе логику какого-либо элемента на странице. При этом этот *объект* берет на себя обязательства *по* генерации собственного *HTML*-представления, обработке параметров HTTP-запроса и другие *операции*, связанные с этим элементом. Таким образом, можно сказать, что элемент управления *ASP.NET Web* Forms является аналогом элемента управления настольного приложения. Платформа *ASP.NET* уже содержит ряд стандартных, наиболее часто используемых элементов управления, таких как кнопка, *поле* ввода, переключатели и т.д.

Способы проверки данных в ASP.NET MVC

Использование валидаторов в приложениях на базе *ASP.NET* MVC Framework невозможно, поскольку модель разработки на платформе *ASP.NET* MVC Framework предполагает иные подходы и не позволяет использовать *элементы управления*.

Валидация – это процесс проверки данных на соответствие различным критериям. При разработке любого приложения в большинстве случаев разработчику приходится иметь дело с обработкой данных, которые ввел *пользователь* в соответствующие поля на форме. *По* разным причинам *пользователь* может вводить некорректные данные.

Процесс валидации *ASP.NET* MVC Framework заключается в следующем. При обработке действия контроллера, *контроллер* определяет какие именно значения не являются корректными и почему. После этого *контроллер* агрегирует информацию об ошибках, которая впоследствии передается представлению. *Представление* в процессе генерации пользовательского интерфейса имеет возможность сообщить пользователю об ошибках на странице.

Для хранения информации об ошибках в базовом классе контроллера определен *объект* `ModelState`. Используя этот *объект*,*контроллер* имеет возможность передать представлению информацию об ошибках на странице.

Если при обработке запроса *контроллер* обнаруживает ошибку, он должен вызвать метод `AddModelError` объекта `ModelState`, в котором указать имя поля, содержащее ошибку и комментарии, которые должны отобразиться пользователю. После выполнения проверки корректности значений для всех параметров, *контроллер* должен проверить свойство `IsValid` объекта `ModelState`. Если *объект* `ModelState` содержит хотя бы одно *сообщение об ошибке*, свойство `IsValid` будет принимать *значение* "`false`"; в противном случае свойство "`IsValid`" примет *значение* "`true`".

Для отображения информации об ошибке в рамках представления используются вспомогательные методы `ValidationMessage` и`ValidationSummary`.

Метод `ValidationSummary` отображает полный *список* ошибок, которые допустил *пользователь* при вводе информации. Данный метод не содержит параметров и использует *объект* `ModelState` для получения информации об ошибках на странице.

Если в какой-то части страницы требуется отобразить информацию об ошибке для конкретного поля, то следует воспользоваться методом `ValidationMessage`. Этот метод принимает в качестве параметра имя поля, для которого необходимо отобразить информацию об ошибке. Имя определяется при добавлении информации об ошибке в *объект* `ModelState` посредством метода`AddModelError`. Первый *параметр* метода `AddModelError` содержит это имя. Если при вызове метода `ValidationMessage`больше никаких параметров не задано, то в качестве сообщения об ошибке будет использоваться сообщение, заданное при вызове метода `AddModelError` объекта `ModelState`. Можно явно указать *сообщение об ошибке* при вызове метода `ValidationMessage`, определив его во втором параметре этого метода.

Логика работы метода `ValidationMessage` определена следующим образом. Если для поля, для которого вызывается этот метод есть *информация* об ошибке в объекте `ModelState`, то в месте вызова метода `ValidationMessage` появляется *информация* об ошибке; если *поле* не содержит ошибок (*объект* `ModelState` не содежит информации об ошибках для этого поля), то в месте вызова метода `ValidationMessage` не появляется никакого сообщения.

Рассмотрим пример работы с `ModelState`. Пусть имеется класс модели для описания деловой встречи:

```
public class Appointment
```

```
{
    public string ClientName { get; set; }
    public DateTime AppointmentDate { get; set; }
}
```

Создадим HTML-форму и контроллер для ввода данных о встрече.

```
<%@ Page Language="C#" Inherits="System.Web.Mvc.ViewPage" %>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Назначение встречи</title>
</head>
  <body>
    <h1>Назначение встречи</h1>
    <% using (Html.BeginForm()) { %>
    <p>Имя клиента: <%= Html.TextBox("appt.ClientName") %> </p>
    <p>Дата встречи: <%=Html.TextBox("appt.AppointmentDate",
                        DateTime.Now.ToShortDateString()) %>
</p>
    <p><%= Html.CheckBox("acceptsTerms") %>
    <label for="acceptsTerms"> Данные подтверждаю </label> </p>
    <input type="submit" value="Сохранить" />
    <% } %>
  </body>
</html>

public class BookingController : Controller
{
    [AcceptVerbs(HttpVerbs.Get)]
    public ActionResult MakeBooking()
    {
```

```
            return View();
        }
    }
```

Действие `MakeBooking()` показывает пустую форму. Так как на форме используется `BeginForm()` без параметров, при нажатии кнопки «Сохранить» форма будет отправлена действию `MakeBooking()`. Чтобы корректно обработать отправку формы, добавим в контроллер `BookingController` перегруженную версию `MakeBooking()`, указав другое значение в атрибуте `[AcceptVerbs]`:

```
    [AcceptVerbs(HttpVerbs.Post)]
    public ActionResult MakeBooking(Appointment appt, bool
acceptsTerms)
    {
        if (string.IsNullOrEmpty(appt.ClientName))
        ModelState.AddModelError("appt.ClientName", "Введите имя");
        if (ModelState.IsValidField("appt.AppointmentDate"))
        {
            // Формат даты правильный,
            // но нужно проверить на логическую корректность
            if (appt.AppointmentDate < DateTime.Now.Date)
                ModelState.AddModelError("appt.AppointmentDate",
                                         "Эта дата уже прошла");
        }
        if (!acceptsTerms)
            ModelState.AddModelError("acceptsTerms",
                                     "Нужно подтверждение");
        if (ModelState.IsValid)
        {
            // Здесь надо добавить сохранения инфо в БД
            // Показывает представление о том, что все ОК
            return View("Completed", appt);
        }
        else
```

```
            // Показываем форму, чтобы пользователь исправил ошибки
            return View();
        }
```

В перегруженной версии метода `MakeBooking()` информация о введенных данных проверяется, а найденные ошибки записываются в хранилище `ModelState`. Если ошибки есть (`ModelState.IsValid==false`), показывается представление с формой ввода. Чтобы пользователь увидел информацию об ошибках, нужно добавить вывод этой информации в код HTML-страницы. Для этого можно использовать метод расширения `Html.ValidationSummary()`.

```
<h1>Назначение встречи</h1>
<% using (Html.BeginForm()) { %>
<%= Html.ValidationSummary() %>
```

Вместо вывода суммарного описания всех ошибок при помощи `Html.ValidationSummary()`, можно печатать точечные сообщения об ошибках, используя `Html.ValidationMessage()`:

```
<p>Имя клиента: <%= Html.TextBox("appt.ClientName") %>
                <%=
Html.ValidationMessage("appt.ClientName")%></p>
```

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

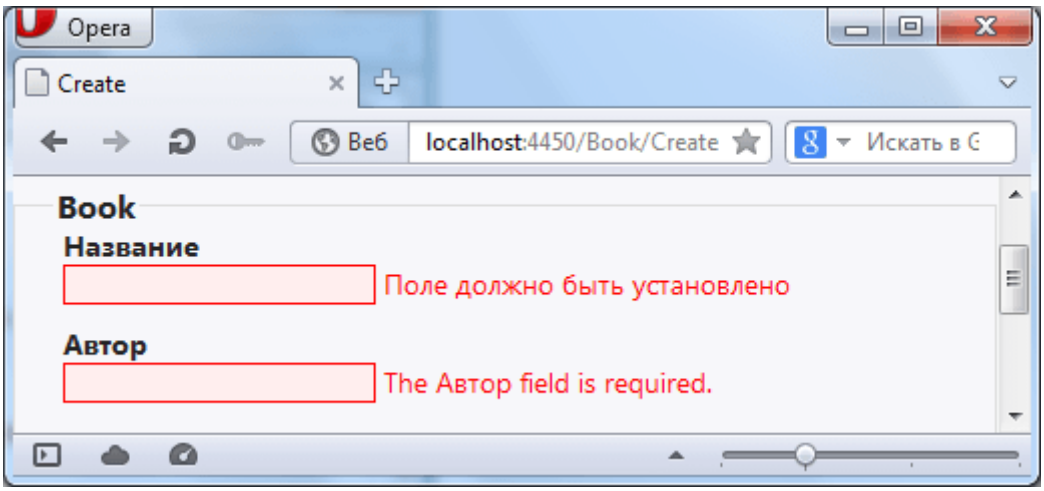
Атрибут Required

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

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

```
1 public class Book
2 {
3     [ScaffoldColumn(false)]
      public virtual int Id { get; set; }
4
5     [Required (ErrorMessage="Поле должно быть установлено")]
6     [Display(Name = "Название")]
7     public virtual string Name { get; set; }
```

```
8
9       [Required]
10      [Display(Name = "Автор")]
11      public virtual string Author { get; set; }
12
13      [Required]
14      [Display(Name = "Год")]
15      public virtual int Year { get; set; }
    }
16
17
```



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

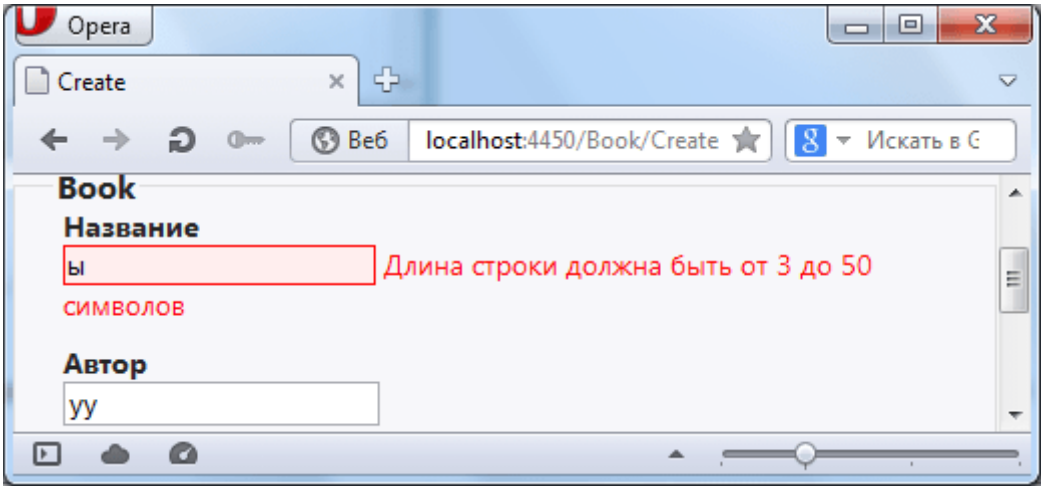
Атрибут StringLength

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

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

```
1  public class Book
2  {
3       [ScaffoldColumn(false)]
4       public virtual int Id { get; set; }
```

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



Атрибут RegularExpression

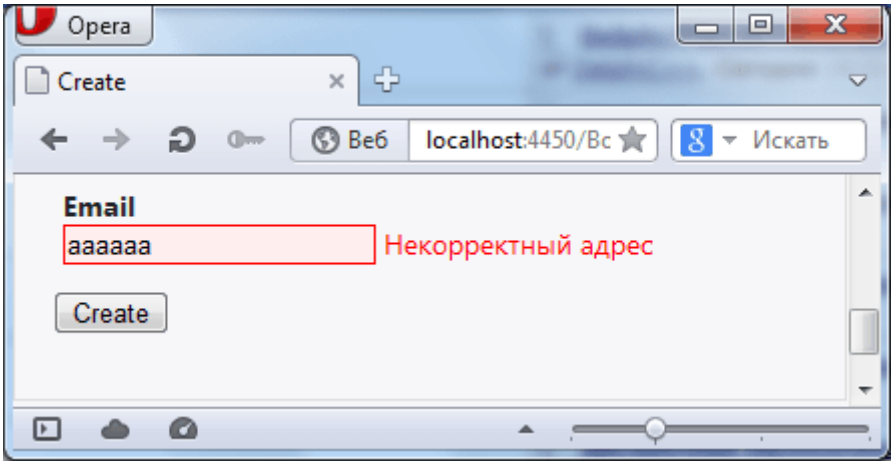
Применение данного атрибута предполагает, что вводимое значение должно соответствовать указанному в этом атрибуте регулярному выражению.

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

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

```
2public virtual string Email { get; set; }
```

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



Атрибут Range

Атрибут Range определяет минимальные и максимальные ограничения для числовых данных.

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

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

```
1[Range(typeof(decimal), "0.00", "49.99")]
2public 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.

```
1[Remote("CheckUserName", "Account")]
2public 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.

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

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

Атрибут Compare

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

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