2) Обработка ошибок в ASP.NET // Представления. Движки представлений. HtmlHelper. Движок Razor

Обработка ошибок в ASP.NET

Если в процессе работы страницы было сгенерировано необработанное исключение, ASP.NET показывает «желтый экран смерти» - стандартную страницу, содержащую информацию об ошибке.

ASP.NET предлагает разработчикам две глобальные точки перехвата программной обработки исключений. В базовом классе `Page` определено событие `Error`, которое можно обрабатывать, перехватывая любые необработанные исключения, выбрасываемые в ходе выполнения страницы. Одноименное событие имеется и у класса `HttpApplication`, оно служит для перехвата необработанных исключений на уровне приложения.

Исключения уровня страницы называются внутренними. Информация о них находится в свойстве **InnerException** класса **Exception**. Необработанные внутренние исключения вызывают исключения уровня приложения – **HttpUnhandledException**.

Для генерации внутреннего исключения на уровне приложения необходимо использовать следующий конструктор:

***Exception exc = new myException();** // создание внутр. пользователь-го исключения*

***exc.Data.Add("message",** "Внутренняя ошибка…**");** // информация об исключении*

***throw new Exception(**"Ошибка приложения …**", exc);** // генерация исключения приложения*

Рассмотрим пример обработчика события `Error` на странице.

```
protected void Page_Error(object sender, EventArgs e)
{
    // перехватываем ошибку
    var ex = Server.GetLastError();
    // выбираем в зависимости от её типа страницу и делаем переход
    if (ex is NotImplementedException)
    {
        Server.Transfer("~/errorpages/notimplemented.aspx");
    }
    else
    {
        Server.Transfer("~/errorpages/apperror.aspx");
```

```
    }
}
```

Объект, представляющий исключение, можно получить с помощью метода `GetLastError()` объекта `Server`. В обработчике события `Error` можно передать управление определенной странице и таким образом вывести персонализированное сообщение, индивидуальное для конкретной ошибки. При этом URL в адресной строке браузера не изменится, поскольку переключение страниц будет выполнено на сервере. Благодаря использованию метода `Server.Transfer()` информация об исключении сохранится, и страница с сообщением об ошибке сама сможет вызвать метод `GetLastError()`, чтобы вывести для пользователя максимально подробные сведения. После того как исключение будет полностью обработано, необходимо удалить объект-ошибку, вызвав метод `Server.ClearError()`.

Обработчик события `Error` страницы перехватывает лишь ошибки, происходящие на этой странице. Если вы решили, что для всех страниц приложения будет использоваться один и тот же обработчик ошибок, то лучше создать глобальный обработчик ошибок на уровне приложения. Такой обработчик реализуется в точности так же, как обработчик ошибок страницы. Добавьте в приложение файл `global.asax` и заполните кодом предопределенную заглушку `Application_Error`.

Для работы с ошибками можно использовать раздел `<customErrors>` файла `web.config` приложения.

```
<system.web>
    <customErrors mode="RemoteOnly" />
</system.web>
```

Обязательный атрибут `mode` определяет, будет ли вывод пользовательских сообщений об ошибках включен, отключен или включен только для удаленных клиентов. По умолчанию он имеет значение `RemoteOnly`, при котором удаленные пользователи видят стандартную страницу с минимально информативным сообщением об ошибке, а локальные пользователи при этом получают сообщения ASP.NET с детальными описаниями ошибок.

Для того чтобы выводить более профессиональные и дружественные пользователю сообщения, которые были бы согласованы с общим интерфейсом сайта, необходимо включить в файл `web.config` такие установки:

```
<system.web>
    <customErrors mode="RemoteOnly"
                  defaultRedirect="GenericError.aspx" />
</system.web>
```

После этого, какой бы ни была ошибка, ASP.NET станет переадресовывать пользователя на страницу GenericError.aspx, содержимое и структура которой всецело определяются разработчиком. Это происходит благодаря необязательному атрибуту defaultRedirect, в котором задается страница с сообщением об ошибке. Если атрибут mode установлен в On, и локальные, и удаленные пользователи перенаправляются на стандартную страницу с сообщением об ошибке. Если же этот атрибут установлен в RemoteOnly, удаленные пользователи перенаправляются на указанную вами страницу, а локальные (которыми обычно являются разработчики) - на выводимую по умолчанию.

ASP.NET позволяет задать отдельную страницу для каждой из ошибок HTTP. Соответствие между страницами с сообщениями об ошибках и кодами состояния HTTP также определяется в web.config. Для раздела <customErrors> поддерживается внутренний тэг <error>, который можно использовать для связывания кодов состояния HTTP с пользовательскими страницами ошибок.

```
<customErrors mode="RemoteOnly"
              defaultRedirect="GenericError.aspx" >
  <error statusCode="404" redirect="Error404.aspx" />
  <error statusCode="500" redirect="Error500.aspx" />
</customErrors>
```

В атрибуте statusCode этого тэга задается код ошибки HTTP, а в атрибуте redirect – страница, куда в случае возникновения такой ошибки должен быть перенаправлен пользователь.

Если для всех страниц приложения алгоритм обработки ошибок один и тот же, то его следует реализовывать на уровне приложения. Такой обработчик будет перехватывать все исключения которые не были обработаны на страницах приложения при условии, что в разделе **<customErrors>** файла **web.config** атрибут **mode = "Off "** или отсутствует **defaultRedirect**.

Пример обработчика в файле **Global.asax**:

```
protected void Application_Error(object sender, EventArgs e)
  {
  // определяем необработанное исключение HttpUnhandledException
  Exception ex = Server.GetLastError();
  if (ex.InnerException != null) // если оно содержит внутренние исключения,
  {
    // анализируем внутренние исключения:
    if (ex.InnerException is NotImplementedExeption)
          Server.Transfer("~/errorpages/notImplementedExeption.htm");
    else
          Server.Transfer("~/errorpages/appError.htm");
  }
```

```
if (ex is HttpException) { // Ошибки уровня HTTP-запросов для .aspx-ресурсов (не .html)
        Server.Transfer("~/errorpages/HttpException.htm");
}
// удаляем ошибку
Server.ClearError();
 }
```

Представления. Движки представлений. HtmlHelper. Движок Razor

Представления отображают данные модели. По умолчанию в ASP.NET MVC применяется «движок» представлений Web Forms View Engine, основанный на элементах классического ASP.NET – aspx-страницах и ascx-элементах. Как отдельные проекты, для ASP.NET MVC существуют «движки» представления NVelocity, Brail, Spark, NHaml, Razor.

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.

Основы Web Forms View Engine

В Web Forms View Engine отдельное представление является aspx-страницей[1]. Следуя конвенции именования, для контроллера `XyzController` и действия `Action()` представление должно называться `Action.aspx` и располагаться в папке `~/Views/Xyz` или в папке `~/Views/Shared`. Базовым классом для представлений является `System.Web.Mvc.ViewPage` или универсальный класс `System.Web.Mvc.ViewPage<T>`. В первом случае представление называется *нетипизированным* (или *слаботипизированным*), во втором - *строготипизированным*. Предполагается, что параметром универсального класса является один из типов модели.

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

This is a <i>very</i> simple view.
```

Представление может иметь ссылку на эталонную страницу. В отличие от страниц классического ASP.NET, представления лишены файла Code Behind. Это связано с отказом от событийной модели и традиционных элементов управления. В ASP.NET MVC разработчик строит внешний вид страницы, используя чистый HTML и вкрапления серверного кода. Для облегчения данного процесса имеется набор методов, обеспечивающий вывод стандартных HTML-элементов.

Рассмотрим пример. Пусть имеется класс `Person`:

```
public class Person

{

    public string Name { get; set; }
```

[1] Существует понятие *частичного представления* – ascx-элемент управления.

```
        public int? Age { get; set; }
        public IEnumerable<Person> Children { get; set; }
    }
```

Создадим строготипизированное представление для вывода информации об объекте класса `Person` (обратите внимание на вкрапления серверного кода):

```
<%@ Page Language="C#"
         Inherits="System.Web.Mvc.ViewPage<Person>" %>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title><%= Model.Name %></title>
</head>
  <body>
    <h1>Information about <%= Model.Name %></h1>
    <p><%= Model.Name %> is <%= Model.Age %> years old.</p>
    <h3>Children:</h3>
    <ul>
        <% foreach (var child in Model.Children) { %>
        <li>
            <b> <%= child.Name %></b>, age <%= child.Age %>
        </li>
        <% } %>
    </ul>
  </body>
</html>
```

Приведённый пример демонстрирует, как представление может получить данные модели. В строготипизированном представлении данные доступны через типизированное свойство `Model`. В любом представлении есть коллекция `ViewData` - слаботипизированный словарь с дополнительным свойством `Model`.

Методы для вывода HTML-элементов

В классе `System.Web.Mvc.ViewPage` определено свойство `Html`, имеющее тип `System.Web.Mvc.HtmlHelper`, и свойство `Url` типа `System.Web.Mvc.UrlHelper`[2]. При импортировании в проект пространства имён `System.Web.Mvc.Html` у класса `HtmlHelper` появляется набор методов расширения для генерирования элементов HTML.

Первую группу методов расширения составляют методы для элементов ввода – флажков, текстовых полей, паролей. В примере кода демонстрируется использование всех методов данной группы.

---

[2] В строготипизированном представлении это типы `HtmlHelper<T>` и `UrlHelper<T>`.

```
<%@ Page Language="C#" Inherits="System.Web.Mvc.ViewPage" %>
<html>
  <body>
    <p><%= Html.CheckBox("Check Box ", true) %></p>
    <p><%= Html.Hidden("Скрытое поле", "info") %></p>
    <p><%= Html.RadioButton("Radio button", "info", true) %></p>
    <p><%= Html.Password("Пароль", "info") %></p>
    <p><%= Html.TextArea("Область ввода", "info", 5, 20,
null)%></p>
    <p><%= Html.TextBox("Поле ввода", "info", true) %></p>
  </body>
</html>
```

Первый параметр методов из описываемой группы задаёт значение HTML-атрибутов `id` и `name`. Для вычисления атрибута `value` применяется следующий алгоритм:

1. Если задано `ViewData.ModelState["name"].Value.RawValue`, где *name* – это имя HTML-элемента, то это значение записывается в атрибут `value`. Словарь `ModelState` хранит данные формы, переданные в предыдущем запросе.
2. Если задан второй строковый параметр методов расширения, то он записывается в атрибут `value`. Иначе переходим к шагу 3.
3. В атрибут `value` записывается результат вызова функции `ViewData.Eval("name")`. Функция `Eval()` сканирует словарь `ViewData`, а затем свойства объекта `Model` для нахождения элемента с указанным именем.

Вторая группа методов предназначена для вывода ссылок и построения URL. Это методы расширения класса `HtmlHelper` - `ActionLink()` и `RouteLink()`, и методы расширения класса `UrlHelper` - `Content()`, `Action()` и `RouteUrl()`.

Метод `ActionLink()` генерирует тег `<a>` и имеет несколько перегруженных версий. В простейшем варианте указывается отображаемый текст и имя действия того же контроллера, который вызвал отображение представления.

```
<%= Html.ActionLink("Link text", "Action") %>
```

Можно явно задать имя контроллера (без суффикса `Controller`) и (или) передать набор значений для параметров маршрута, определяющего контроллер.

```
<%= Html.ActionLink("Text", "Action", "Home") %>
<%= Html.ActionLink("Text", "Action", new { controller = "Home"
})%>
<%-- если параметра p нет в шаблоне URL,
```

```
        этот вызов транислируется в ссылку на /Home/Action?p=val --
%>

    <%= Html.ActionLink("Link text", "Action",
                        new { controller = "Home", p = "val" })%>
```

Метод `ActionLink()` также можно использовать для создания обычной веб-ссылки.

```
    <%-- транислируется в https://www.example.com/Home/About#anchor
--%>

    <%= Html.ActionLink("Hi", "About", "Home", "https",
                        "www.example.com", "anchor", new { },
null)%>
```

Метод `RouteLink()` генерирует тег `<a>` для маршрута. В качестве параметров метод принимает текст ссылки, имя маршрута в таблице маршрутизации, набор параметров маршрута. Методы расширения класса `UrlHelper` похожи на методы `ActionLink()` и `RouteLink()`, но генерируют не тег `<a>`, а простую строку, описывающую ссылку.

Два метода расширения предназначены для вывода выпадающего списка и списка выбора. Это методы `DropDownList()` и `ListBox()`. Данные методы используют параметры типов `MultySelectList` и `SelectList`. Эти классы создают списки выбора на основе коллекций.

```
    <% var list = new SelectList(regionsData,  // коллекция объектов
                                 "RegionID",   // поле объекта для
value
                                 "RegionName", // поле объекта для
text
                                 3);           // выбранное значение
%>

    <%= Html.DropDownList("List", list, "Choose") %>

    <%= Html.ListBox("List", new MultiSelectList(new [] {"A", "B"}))
%>
```

Метод расширения `Encode()` позволяет преобразовать текст, чтобы он не воспринимался браузерами как HTML. Например, символ < представляется в виде `<` и так далее.

```
    <%= Html.Encode("I'm <b>\"HTML\"-encoded</b>") %>
```

Построение HTML-форм ввода

Для получения информации элементы ввода на HTML-странице должны быть сгруппированы в форму. Тег и атрибуты формы можно задать вручную или применить методы расширения класса `HtmlHelper` `BeginForm()` и `EndForm()`. Метод

`BeginForm()` можно использовать с инструкцией `using`, что ведёт к автоматической генерации закрывающего тега формы.

```
<% Html.BeginForm("MyAction", "MyController"); %>
    здесь располагаются элементы ввода и кнопка submit
<% Html.EndForm(); %>


<%-- эквивалентный синтаксис --%>
<% using (Html.BeginForm("MyAction", "MyController")) { %>
    здесь располагаются элементы ввода и кнопка submit
<% } %>
```

Достоинство метода `BeginForm()` заключается в том, что при помощи его параметров легко сформировать у формы правильный атрибут `action`. В простейшем варианте `BeginForm()` используется без параметров – в этом случае запрос отправляется тому же действию, которое вызвало представление. Можно указать имя действия и имя контроллера (см. пример выше), параметры маршрута, метод отправки формы (POST или GET) и имена и значения произвольных HTML-атрибутов:

```
<% Html.BeginForm("MyAction", "MyController",
                  new { param = "val" }, FormMethod.Get); %>
```

Строчные хелперы

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

```
@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)
```

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. Итак, создадим в проекте новую папку Helpers и добавим в нее новый класс ListHelper:

```
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

Работа с частичными представлениями

Web Forms View Engine поддерживает *частичные представления*, созданные как ascx-элементы управления. Базовым классом для частичных представлений является `System.Web.Mvc.ViewUserControl` или универсальный класс `System.Web.Mvc.ViewUserControl<T>`.

Если частичное представление создано, оно выводится на страницу при помощи метода расширения `Html.RenderPartial()`. Метод принимает в качестве параметра имя частичного представления и, возможно, пользовательский объект. Этот объект доступен в частичном представлении через свойство `ViewData.Model`.

Опишем частичное представление `PersonInfo.ascx` для отображения информации объекта класса `Person`:

```
<%@ Control Language="C#"
            Inherits="System.Web.Mvc.ViewUserControl<Person>" %>
Возраст <%= Model.Name %> составляет <%= Model.Age %> лет
```

Используем `PersonInfo.ascx`, чтобы вывести список объектов:

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

```
        <body>
          Списком людей:
          <ul>
              <% foreach (var p in (IEnumerable)ViewData["people"]) {
%>
              <li>
                  <% Html.RenderPartial("PersonInfo", p); %>
              </li>
              <% } %>
          </ul>
        </body>
    </html>
```

Движок Razor

ASP.NET MVC всегда поддерживал концепцию “движка представлений”, собственно он представляет из себя заменяемые модули, которые реализуют выбор различного синтаксиса шаблона. Сегодня, стандартный движок представлений для ASP.NET MVC использует аналогичные файлы, что и ASP.NET Web Forms — .aspx/.ascx/.master.

Движок представлений оптимизирован под генерацию HTML-кода, фокусируясь на коде шаблона. Кодовое имя для данного движка – “Razor”.

Основы синтаксиса Razor

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

Например, переход к выражению кода:

```
<p>@b.Name</p>
```

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

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

```
<h3>@DateTime.Now.ToShortTimeString()</h3>
```

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

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

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

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

Создание типов, методов, объектов и переменных. Блок обозначенный как *@functions{...}* дает возможность создавать любой синтаксически корректный код на языке программирования *C#* равносильно созданию кода в файле с расширением .cs. Код в блоке *@functions{...}* воспринимается как составляющая класса с именем идентичным названию файла .cshtml. Разместив файл .cshtml папке App_Code и создав переменные, методы и классы с модификатором доступа public можно получить доступ к коду с любой страницы сайта. Например, вы создали файл Utility.cshtml, то код будет доступен на других страницах через @Utility.название_метода()

```
@functions{

    // Новый класс
    static class MyClass
    {
        /// <summary>
        /// Описание метода для документирования
        /// </summary>
        /// <returns>описание возвращаемого значения</returns>
        public static string GetTicks()
        {
            return DateTime.Now.Ticks.ToString();
        }

        private static string s = "123456";
    }


    // Функция с возвратом
    public string Function1()
    {
        return "Привет!!!";
```

```
    }

    // Функция без возврата
    void Function2()
    {
        // Задержка для разности результатов при многократных
        // последовательных вызовах функции GetTicks()
        Thread.Sleep(1);
    }

}
```

При необходимости вывода текста, содержащего зарезервированные слова и названия объектов предназначен показанный ниже синтаксис:

```
@{
    // В html-тегах зарезервированные слова отображаются как
простой текст
    <p>
        int double string new null if for  class delegate
Request Response DateTime
    </p>

    // Следующий способ преобразования ключевых слов в простой
текст
    @:int double string new null if for  class delegate
Request Response DateTime

    // Использование не отображаемых тегов <text>
    <text>
        int double string new null if for  class delegate
Request Response DateTime
    </text>

}
```