## 9) Поддержка AJAX в ASP.NET Web Forms // Контроллеры и действия MVC

### Поддержка AJAX в ASP.NET Web Forms

В этом параграфе пойдет речь о некоторых стандартных элементах управления ASP.NET, с помощью которых сравнительно легко интегрировать поддержку AJAX для страниц.

**1. Элемент управления `ScriptManager`.**

Элемент управления `ScriptManager` служит для доставки клиенту JavaScript-файлов. В простейшей ситуации элемент размещается на странице без дополнительной настройки и обеспечивает доставку скриптов, реализующих AJAX-функционал. Однако данный элемент обладает рядом дополнительных возможностей. Использовав его свойство `Scripts` и вложенные элементы `ScriptReference`, можно передать клиенту любой js-файл, в том числе и внедренный в некоторую сборку в качестве ресурса. В следующем примере клиент получает скрипты `sc1.js` и `sc2.js`, а также внедренный скрипт из сборки `Demo`.

```
<asp:ScriptManager ID="sm" runat="server">
  <Scripts>
    <asp:ScriptReference Path="~/scripts/sc1.js" />
    <asp:ScriptReference Path="~/scripts/sc2.js" />
    <asp:ScriptReference  Assembly="Demo" Name="Demo.script.js" />
  </Scripts>
</asp:ScriptManager>
```

Элемент управления `ScriptManager` также позволяет создать прокси-функции на JavaScript для асинхронного доступа к веб-службе ASP.NET. Веб-служба, которую планируется вызывать асинхронно, должна быть помечена дополнительным атрибутом `[ScriptService]`.

```
<%@ WebService Language="C#" Class="NameSp.Serv" %>

using System;
using System.Web.Services;
using System.Web.Script.Services;

namespace WebProject
{
    [ScriptService]
    [WebService]
```

```
    public class Serv
    {
        [WebMethod]
        public int GetCount()
        {
            return 23;
        }
    }
}
```

Для установки ссылки на службу и создания прокси на стороне клиента используется вложенный элемент `ServiceReference`:

```
<asp:ScriptManager ID="sm" runat="server">
  <Services>
    <asp:ServiceReference Path="~/serv.asmx" />
  </Services>
</asp:ScriptManager>
```

Имя клиентской прокси-функции формируется из имени пространства имен, имени класса веб-службы и имени метода службы. Функции передается необходимый список параметров и три параметра дополнительно. Первый из дополнительных параметров указывает на callback-функцию, вызываемую в случае удачного ответа службы, второй – на функцию, вызываемую в случае неудачи, а третий параметр – это произвольный объект-контекст вызова.

```
<script>
    WebProject.Serv.GetCount(onSuccess, onFailed, 10);

    function onSuccess(result, context, methodName) {...}

</script>
```

**2. Элемент управления `UpdatePanel`.**

Контейнерный элемент `UpdatePanel` реализует возможность частичного изменения разметки на aspx-страницах. Такие страницы почти не отличаются от обычных, если не считать того, что в них включается элемент `ScriptManager` и один или несколько элементов `UpdatePanel`. Каждый экземпляр элемента `UpdatePanel` задает области страницы, которые могут обновляться независимо. Элементы `UpdatePanel` могут

размещаться внутри пользовательских элементов, на эталонной странице (Master Page) и на страницах содержимого.

Приведем пример фрагмента страницы, использующей `UpdatePanel`:

```
<form id="form1" runat="server">
  <div>
    <asp:ScriptManager ID="ScriptManager1" runat="server" />
    <asp:UpdatePanel ID="UpdatePanel1" runat="server">
      <ContentTemplate>
        <asp:Textbox runat="server" id="tbx" />
        <asp:Button runat="server" id="btn" OnClick="btn_Click" />
      </ContentTemplate>
    </asp:UpdatePanel>
  </div>
</form>
```

Следует обратить внимание на присутствие на странице элемента `ScriptManager` (он должен быть размещен перед первым `UpdatePanel`). Также обратите внимание на задание серверного обработчика события вложенной кнопки. Хотя происходит только частичное обновление страницы, события страницы срабатывают в обычном порядке (правда, не все, а только `Page_Init`, `Page_Load`, `Page_PreRender` и `Page_Unload`).

Основные свойства элемента `UpdatePanel` приведены в таблице 8.

Таблица 8

Свойства элемента `UpdatePanel`

| Имя | Описание |
|---|---|
| `ContentTemplate` | Определяет шаблон, то есть исходное содержимое `UpdatePanel` |
| `IsUpdating` | Булево свойство; указывает, находится ли панель в процессе обновления по асинхронному возврату данных |
| `Mode` | Определяет, при каких условиях происходит обновление панели. Допустимые значения (перечисление `UpdatePanelMode`): `Always` (по умолчанию) - панель обновляется для каждого возврата данных, инициированного страницей, `Conditional` – панель обновляется только при срабатывании триггеров или при программном запросе |
| `RenderMode` | Указывает, как генерируется содержимое панели - как встроенный код или в виде блока. Допустимые значения объединены в перечисляемый тип `UpdatePanelRenderMode`. По умолчанию используется режим `Block` |
| `Triggers` | Коллекция объектов-триггеров. Каждый объект предоставляет событие, приводящее к автоматическому обновлению панели |

В предыдущем примере элемент, который являлся причиной асинхронного возврата данных, располагался внутри обновляемой панели. На практике это условие выполняется далеко не всегда. Панель может обновляться по щелчку на любом элементе страницы, который может стать причиной возврата данных. Если элемент находится за пределами панели, произойдет полное обновление страницы - если только элемент не был зарегистрирован в качестве *триггера обновления* для панели. Для каждого элемента `UpdatePanel` можно определить сразу несколько триггеров. При срабатывании любого из них панель автоматически обновляется. Триггеры могут определяться как на декларативном, так и на программном уровне. В первом варианте используется секция `<Triggers>`, а во втором - свойство-коллекция `Triggers`.

ASP.NET AJAX поддерживает два типа триггеров: `AsyncPostBackTrigger` и `PostBackTrigger`. Класс `AsyncPostBackTrigger` позволяет указать идентификатор элемента управления вне `UpdatePanel` и имя события, при наступлении которого панель обновляется. Класс `PostBackTrigger` служит для указания элемента управления внутри `UpdatePanel` для синхронной отправки страницы на сервер.

```
<asp:UpdatePanel ID="UpdatePanel1" runat="server">
    <ContentTemplate> . . . </ContentTemplate>
    <Triggers>
      <asp:AsyncPostBackTrigger ControlID="btn" EventName="Click" />
      <asp:PostBackTrigger ControlID="btnInside" />
    </Triggers>
</asp:UpdatePanel>
```

**3. Элементы управления `Timer` и `UpdateProgress`.**

Элемент управления `Timer` представляет собой таймер, размещенный на клиенте и срабатывающий через указанные промежутки времени. Промежуток задается в миллисекундах в свойстве `Interval`, а срабатывание приводит к генерации события `Tick`. Как правило, данный элемент применяется в сочетании с триггером некой `UpdatePanel` для автоматического обновления области страницы через некие временные промежутки.

```
<asp:UpdatePanel ID="up" runat="server">
  <ContentTemplate>    </ContentTemplate>
  <Triggers>
    <asp:AsyncPostBackTrigger ControlID="tmr" EventName="Tick" />
  </Triggers>
</asp:UpdatePanel>
<asp:Timer ID="tmr" runat="server" Interval="1000" />
```

Элемент `UpdateProgress` обеспечивает обратную связь в браузере во время обновления одного или нескольких элементов `UpdatePanel`. Элемент `UpdateProgress` можно разместить в любом месте страницы, определяя его стиль и позицию при помощи CSS. Страница может содержать не более одного элемента `UpdateProgress`, который обслуживает все обновляемые панели на странице. Связь между элементом `UpdatePanel` и `UpdateProgress` устанавливается автоматически. Элемент `UpdateProgress` обладает единственным свойством `ProgressTemplate`. Свойство определяет шаблон с разметкой, которая должна отображаться во время обновления панели. Шаблон может содержать произвольную комбинацию элементов. Впрочем, обычно в него помещается небольшой текстовый фрагмент и анимированное изображение в формате GIF.

## Контроллеры и действия MVC

В шаблоне MVC контроллеры отвечают за *логику приложения*, которая включает получение пользовательского ввода, выполнение команд над моделью и управление представлениями. В объектно-ориентированных языках контроллер обычно представлен как класс, методы которого называют *действиями контроллера* (*actions*).

В ASP.NET MVC класс контроллера должен реализовывать интерфейс `IController`. Это единственное строгое требование к контроллеру. Ниже дано описание интерфейса `IController`, а также пример примитивного контроллера.

```
public interface IController

{

    void Execute(RequestContext requestContext);

}


public class HelloWorldController : IController

{

    public void Execute(RequestContext requestContext)

    {

        requestContext.HttpContext.Response.Write("Hello, world!");

    }

}
```

При практическом использовании ASP.NET MVC создание контроллера только на основе `IController` не применяется. Обычно пользовательские контроллеры наследуются от класса `System.Web.Mvc.Controller` (дополнительно применяется соглашение об именовании – имя класса-контроллера заканчивается на `Controller`). Методы пользовательского контроллера становятся действиями и доступны из веб, если выполняются все следующие условия:

- Метод объявляен как `public` и не является статическим;
- Метод не объявлен как универсальный;
- Метод не помечен атрибутом `[NonAction]`;

- У метода нет перегруженных версий, за исключением случаев, когда эти версии помечены атрибутами [NonAction] или [AcceptVerbs].

### *Получение входных данных*

Обычно действию контроллера требуются для выполнения некие входные данные. В ASP.NET MVC существуют три основных способа для получения контроллером такой информации.

Первый способ предполагает извлечение входных данных из различных *контекстных объектов*, доступных в контроллере при наследовании от System.Web.Mvc.Controller. В табл. 2 перечислены основные контекстные объекты.

Таблица 2

Основные контекстные объекты, доступные в контроллере

| Свойство для доступа | Его тип | Описание |
|---|---|---|
| Request.QueryString | NameValueCollection | Переменные, переданные в GET-запросе |
| Request.Form | NameValueCollection | Переменные, переданные в POST-запросе |
| Request.Cookies | HttpCookieCollection | Cookies, посланные браузером |
| Request.HttpMethod | string | HTTP-метод запроса (GET, POST,…) |
| Request.Headers | NameValueCollection | Все HTTP-заголовки запроса |
| Request.Url | string | Запрашиваемый URL |
| RouteData.Route | RouteBase | Объект из таблицы маршрутизации, соответствующий запросу |
| RouteData.Values | RouteValueDictionary | Словарь параметров маршрута и их значений |
| HttpContext.Cache | Cache | Кеш приложения |
| HttpContext.Session | HttpSessionStateBase | Информация сессии пользователя |
| User | IPrincipal | Информация об аутентифицированном пользователе |
| TempData | TempDataDictionary | Данные, сохранённые при обработке предыдущего запроса |

Второй способ получения входных данных – описать и использовать параметры метода-действия. Каркас ASP.NET MVC пытается самостоятельно заполнить значения параметров, используя данные контекстных объектов контроллера. Для каждого из параметров выполняется поиск значения по имени параметра без учета регистра. В случае неудачи или ошибки приведения типов, параметры ссылочного типа получают значение null, а для параметров типов-значений генерируется исключение.

Третий способ извлечения входных данных предполагает использование специального механизма ASP.NET MVC, называемого *привязка к модели* (*model binding*). Вместо атомарных входных параметров привязка к модели позволяет указать единственный параметр-объект. Заполнение атрибутов этого объекта будет выполнено автоматически, подобно тому, как это делалось для примитивных параметров во втором способе. Пусть, например, имеется модельный класс Person:

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

```
    public int? Age { get; set; }
}
```

Можно создать следующий метод-действие для обновления `Person`:

```
public ActionResult Update(string name, int? age)
{
    var person = new Person{ Name = name, Age = age};
    ViewData["Message"] = person.Name + " " + person.Age;
    return View();
}
```

Использование привязки к модели позволяет упростить метод `Update()`:

```
public ActionResult Update(Person person)
{
    ViewData["Message"] = person.Name + " " + person.Age;
    return View();
}
```

Привязку к модели можно запустить принудительно, вызвав метод `UpdateModel()`.

```
public ActionResult Update()
{
    var person = new Person();
    UpdateModel(person);
    ViewData["Message"] = person.Name + " " + person.Age;
    return View();
}
```

### *Генерирование выходных данных*

При использовании контроллеров, унаследованных от `System.Web.Mvc.Controller`, действия возвращают объект для описания результата своей работы. Как правило, используется класс `ActionResult` и его наследники[1], перечисленные в табл. 3.

[1] Метод-действие может возвращать произвольный объект или быть объявленным как `void`. В первом случае на основе результата создаётся объект класса `ContentResult`, во втором – возвращается объект `EmptyResult`.

Таблица 3

Встроенные типы для представления результата действия

| Имя типа | Описание |
|---|---|
| ViewResult | Возвращает представление по умолчанию для контроллера или представление с указанным именем |
| PartialViewResult | Возвращает частичное представление по умолчанию для контроллера или частичное представление с указанным именем |
| RedirectToRouteResult | Вызывает перенаправление (HTTP 302) на указанный метод-действие или маршрут |
| RedirectResult | Вызывает перенаправление (HTTP 302) на указанный URL |
| ContentResult | Возвращает браузеру обычный текст (можно указать значение заголовка content-type) |
| FileResult | Передаёт клиенту двоичные данные (файл, массив байт) |
| JsonResult | Сериализует объект в формате JSON и передаёт клиенту |
| JavaScriptResult | Посылает клиенту фрагмент кода на JavaScript или js-файл |
| HttpUnauthorizedResult | Посылает клиенту HTTP-код 401 для запуска механизма авторизации |
| EmptyResult | Не выполняет никаких действий |

Во фрагменте кода, представленном ниже, показаны примеры использования типов для результата действия. Заметим, что для создания объектов рассматриваемых типов можно использовать как обычный конструктор, так и встроенные методы класса System.Web.Mvc.Controller.

```
public ActionResult Index()
{
    // 1а. Возвращаем представление, используя конструктор
    return new ViewResult { ViewName = "HomePage" };
    // 1б. Возвращаем представление, используем встроенный метод
    return View("HomePage");
    // 1в. Без параметров - возвращаем представление по умолчанию
    return View();

    // 2. Выполняем редирект на действие контроллера
    // (имени контроллера нет, значит действие в том же контроллере)
    return RedirectToAction("Show");

    // 3. Выполняем редирект на указанный маршрут по имени в таблице
    // маршрутизации (можно передать параметры маршрута)
    return RedirectToRoute("MyNameRoute", new { param = "value" });
```

```
        // 4. Выполняем редирект на указанный URL
        return Redirect("http://example.com");

        // 5. Возвращаем обычный текст
        return Content("This is a plain text", "text/plain");

        // 6. Сериализуем некий объект в JSON и возвращаем клиенту
        return Json(obj);

        // 7. Передаём клиенту файл
        return File("somefile.pdf", "application/pdf");
    }
```

Когда результатом действия является представление, возникает задача передачи информации этому представлению. Для этого применимы несколько вариантов. Класс `Controller` имеет коллекцию `ViewData`, представляющую собой слаботипизированный словарь. Эта коллекция доступна и в представлении:

```
// код в контроллере
ViewData["Message"] = "Welcome to ASP.NET MVC!";

<%-- код в представлении --%>
<%= Html.Encode(ViewData["Message"]) %>
```

У `ViewData` есть специальное свойство `Model` (произвольный объект). Свойство можно задать непосредственно или использовать для этого параметр метода `View()`.

```
var person = new Person{ Name = "Mister X", Age = 20};
// первый вариант
ViewData.Model = person;
// второй вариант
return View(person);
```

Ещё один вариант передачи данных представлению - *строготипизированные представления* – будет рассмотрен в дальнейшем.

При выполнении переадресации на маршрут или действие удобно использовать для хранения промежуточных данных особую коллекцию `TempData`, доступную в теле контроллера. `TempData` работает как коллекция `Session`, однако данные, которые туда помещены, сохраняются ровно на один следующий запрос.

```
TempData["myKey"] = person;
```

### *Использование фильтров действий*

Контроллеры и действия могут быть снабжены *фильтрами*, добавляющими дополнительные шаги в процесс обработки запроса. Существуют четыре вида фильтров: фильтры авторизации, действия, результата и исключения. Фильтры авторизации позволяют ограничить доступ к контроллерам или действиям по роли или имени пользователя. Фильтры исключений указывают способ обработки исключения, возникшего при выполнении действия. Фильтры действий и результатов могут определить методы, выполняемые инфраструктурой ASP.NET MVC до или после действия и генерирования результата.

Технически, любой фильтр – это .NET атрибут. Для каждого вида фильтров имеется отдельный интерфейс и базовый класс реализации[2] (табл. 4):

Таблица 4

Интерфейсы и базовые реализации фильтров

| Вид фильтр | Интерфейс | Базовый класс |
|---|---|---|
| Авторизация | `IAuthorizationFilter` | `AuthorizeAttribute` |
| Действие | `IActionFilter` | `ActionFilterAttribute` |
| Результат | `IResultFilter` | `ActionFilterAttribute` |
| Исключение | `IExceptionFilter` | `HandleErrorAttribute` |

Следующий код демонстрирует пример создания простого пользовательского фильтра. В качестве базового класса выбран `ActionFilterAttribute`, реализующий интерфейсы `IActionFilter` и `IResultFilter`.

```
public class ShowMessageAttribute : ActionFilterAttribute
{
    public string Message { get; set; }

    public override void OnActionExecuting(ActionExecutingContext context)
    {
      context.HttpContext.Response.Write("BeforeAction " + Message);
    }
    public override void OnActionExecuted(ActionExecutedContext context)
    {
      context.HttpContext.Response.Write("AfterAction " + Message);
    }
    public override void OnResultExecuting(ResultExecutingContext context)
```

[2] Все классы фильтров наследуются от `FilterAttribute`. У `FilterAttribute` имеется целочисленное свойство `Order` для управления порядком применения фильтров.

```
    {
      context.HttpContext.Response.Write("BeforeResult " + Message);
    }
    public override void OnResultExecuted(ResultExecutedContext context)
    {
      context.HttpContext.Response.Write("AfterResult " + Message);
    }
}
```

Фильтры могут применяться как к отдельному действию, так и ко всему контроллеру. В этом случае считается, что они распространяются на каждое действие контроллера. Альтернативой фильтрам-атрибутам может служить переопределение виртуальных методов класса `System.Web.Mvc.Controller`: `OnActionExecuting()`, `OnActionExecuted()`, `OnResultExecuting()`, `OnResultExecuted()`, `OnAuthorization()` и `OnException()`.

Опишем некоторые встроенные фильтры ASP.NET MVC. Фильтр `[Authorize]` – это фильтр авторизации. При применении фильтра указывается список пользователей и (или) список ролей. Для выполнения действия пользователь должен быть авторизован, указан в списке пользователей и иметь хотя бы одну из указанных ролей.

```
public class MicrosoftController : Controller
{
    [Authorize(Users = "billg, steveb", Roles = "chairman, ceo")]
    public ActionResult BuySmallCompany(string name, double price)
    { . . . }
}
```

Если при выполнении действия было сгенерировано необработанное исключение, фильтр `[HandleError]` позволит вывести представление с описанием ошибки. У `[HandleError]` можно задать такие параметры как тип исключения, имя представления и имя шаблона (master page) для представления. Если не задан тип исключения, обрабатываются все исключения. Если не указано имя представления, используется представление `Error`.

```
[HandleError]
public class HomeController : Controller
{ . . . }
```

Фильтр `[OutputCache]` позволяет кэшировать результат отдельного действия или всех действий контроллера. Свойства фильтра `[OutputCache]` совпадают с параметрами директивы `@OutputCache`, применяемой в классическом ASP.NET.

```
public class StockTradingController : Controller
{
```

```
    [OutputCache(Duration = 60)]
    public ViewResult CurrentRiskSummary()
    { . . . }
}
```

В заключение опишем несколько атрибутов, применяемых к методам контроллера, но не являющихся фильтрами. Атрибут [NonAction] экранирует public-метод так, что этот метод не распознаётся как действие. Атрибут [ActionName] даёт действию имя, с которым сопоставляются параметры URL. Атрибут [AcceptVerbs] позволяет указать HTTP-методы, для которых будет вызываться действие.

```
[AcceptVerbs(HttpVerbs.Get)]
public ActionResult DoSomething() { . . . }


[AcceptVerbs(HttpVerbs.Post)]
public ActionResult DoSomething(int someParam) { . . . }
```