10) Элементы управления для источников данных и проверки данных Web Forms // поддержка AJAX в ASP.NET MVC

Элементы управления для источников данных и проверки данных Web Forms

Элементы управления для представления источников данных – это компоненты ASP.NET, которые служат оболочкой для реальных источников данных и допускают гибкую декларативную настройку. Всего имеется шесть таких элементов управления: `SqlDataSource`, `AccessDataSource`, `ObjectDataSource`, `LinqDataSource`, `XmlDataSource` и `SiteMapDataSource`. Первые четыре предназначены для работы с табличными источниками данных, оставшиеся – для работы с иерархическими данными (XML). `SiteMapDataSource` представляет собой специальный элемент управления, работающий с файлами навигации по сайту, а `AccessDataSource` – специальную реализацию `SqlDataSource` для работы с базами данных Access. Далее будет подробнее рассмотрена работа с `SqlDataSource` и `ObjectDataSource`.

Элемент управления `SqlDataSource`

Элемент управления `SqlDataSource` предназначен для двунаправленного обмена с любым источником данных, к которому есть доступ с использованием управляемых поставщиков ADO.NET. Поддерживается постраничный вывод (для умеющих это делать элементов управления) и сортировка, кэширование и фильтрация. При получении данных можно использовать параметры. Также элемент управления `SqlDataSource` может обновлять данные в источнике данных. Все эти возможности поддерживаются на декларативном уровне. Ограничением является реализация сортировки, постраничного вывода, кэширования и фильтрации только при получении данных с использованием `DataSet`.

Разберём простейший пример использования `SqlDataSource`. Для получения данных достаточно указать у элемента `SqlDataSource` значение свойств `ConnectionString` - строка подключения, `ProviderName` – имя управляемого поставщика и `SelectCommand` – SQL-команда для выборки.

```
<asp:SqlDataSource ID="SqlDS" runat="server" ConnectionString="..."
                   SelectCommand="SELECT * FROM [Performers]"
                   ProviderName="System.Data.SqlClient" />
```

Строку подключения можно задать с использованием элемента из секции `<connectionStrings>` файла `web.config`. При этом можно указать и используемого поставщика, что автоматически освобождает от определения свойства `ProviderName`:

```
<connectionStrings>
  <add name="CatalogCS"
       connectionString="Data Source=(local)\SQLExpress;. . ."
       providerName="System.Data.SqlClient" />
```

```
</connectionStrings>


<asp:SqlDataSource ID="SqlDS" runat="server"
                ConnectionString="<%$ ConnectionStrings:CatalogCS %>"
                SelectCommand="SELECT * FROM [Performers]">
</asp:SqlDataSource>
```

Рассмотрим работу с SqlDataSource на примерах. При наличии опыта использования DataAdapter работа с SqlDataSource не вызовет трудностей. Для задания запроса получения данных используется свойство SelectCommand вместе с сопутствующими свойствами SelectCommandType и SelectParameters. Тип возвращаемого набора данных (и внутренний класс, используемый для хранения данных) определяет свойство DataSourceMode, принимающее значение DataSet или DataReader.

```
<asp:SqlDataSource ID="SqlDS" runat="server"
      ConnectionString="<%$ ConnectionStrings:CatalogCS %>"
      SelectCommand="SelectAll" SelectCommandType="StoredProcedure"
</asp:SqlDataSource>
```

Ознакомимся с заданием параметров для запроса. Параметры выборки данных описываются в коллекции SelectParameters и включаются в текст команды SelectCommand с использованием префикса @.

```
<asp:SqlDataSource ID="SqlDS" runat="server"
       ConnectionString="<%$ ConnectionStrings:CatalogCS %>"
       SelectCommand="SELECT * FROM [Performers] WHERE name=@name">
    <SelectParameters>
        <asp:Parameter Name="name" />
    </SelectParameters>
</asp:SqlDataSource>
```

При таком указании параметра единственный способ его изменить – это использовать программный код. Но SqlDataSource позволяет определить специальные виды параметров, значения которых автоматически вычисляются из самых разных источников – из адресной строки, сессии, профиля пользователя, переменных формы и серверных элементов управления. Например, при получении значения из адресной строки объявление параметра будет таким:

```
<asp:QueryStringParameter Name="name" QueryStringField="getN" />
```

А для получения значения из элемента управления TextBox нужно написать следующий код:

```
<asp:ControlParameter Name="name" ControlID="tbxName"
```

```
                                PropertyName="Text" />
```

При задании параметра можно указать дополнительные свойства: `DefaultValue` - для значения по умолчанию, `Type` – для указания типа данных. Свойство `SqlDataSource.CancelSelectOnNullParameter` определяет, прерывать ли выполнение запроса, если какой-либо из параметров равен `null`, а свойство `Parameter.ConvertEmptyStringToNull` указывает на необходимость конвертации пустых значений параметров в `null`.

Как упоминалось ранее, элемент управления `SqlDataSource` поддерживает кэширование получаемых данных. Кэширование работает только в режиме `DataSet` и включается с помощью свойства `EnableCaching`. Кроме того, у `SqlDataSource` есть свойства для установки политики кэширования – `CacheDuration`, `CacheExpirationPolicy`, `CacheKeyDependency` и `SqlCacheDependency`. `SqlDataSource` кэширует данные с учетом параметров, то есть для приведенного выше примера для каждого значения параметра `name` в кэше будет создана отдельная запись. В некоторых случаях это удобно, но иногда оптимальнее закэшировать в памяти весь набор данных и уже из этого набора выбирать нужные данные фильтрацией. Это можно сделать с помощью свойства `FilterExpression`, задающего выражения для фильтрации, и коллекции `FilterParameters`, задающей значения для фильтра.

Вот пример описания элемента управления `SqlDataSource`, кэширующего получаемый список на одну минуту и использующего фильтрацию по имени:

```
<asp:SqlDataSource ID="SqlDS" runat="server"
                ConnectionString="<%$ ConnectionStrings:CatalogCS %>"
                SelectCommand="SELECT * FROM [Performers]"
                EnableCaching="True" CacheDuration="60"
                FilterExpression="name = '{0}'">
    <FilterParameters>
        <asp:ControlParameter ControlID="tbx" Name="Name"
                              PropertyName="Text" />
    </FilterParameters>
</asp:SqlDataSource>
```

`SqlDataSource` может не только получать данные, но и умеет эти данные обновлять. Как и `DataAdapter`, он использует для этого свойства `UpdateCommand`, `DeleteCommand` и `InsertCommand`, настройка которых в принципе аналогична настройке `SelectCommand`. Для обновления данных `SqlDataSource` может использовать параметризованные запросы или хранимые процедуры

Рассмотрим пример, в котором `GridView` и `SqlDataSource` используются не только для отображения, но и для редактирования данных таблицы `Performers`. Начало описания `SqlDataSource` уже знакомо:

```
<asp:SqlDataSource ID="SqlDS" runat="server"
                ConnectionString="<%$ ConnectionStrings:CatalogCS %>"
                SelectCommand="SELECT * FROM [Performers]"
```

Так как элемент управления GridView поддерживает редактирование и удаление данных, но не поддерживает их добавления, определим только Update и Delete команды:

```
        DeleteCommand="DELETE FROM [Performers] WHERE [id] = @old_id"
        UpdateCommand="UPDATE [Performers] SET [name] = @name,
                    [is_group] = @is_group WHERE [id] = @old_id"
```

Также нужно задать свойства для передачи оригинальных параметров:

```
        ConflictDetection="OverwriteChanges"
        OldValuesParameterFormatString="old_{0}" >
```

Теперь в SqlDataSource осталось только описать коллекции параметров для команд обновления и удаления:

```
<DeleteParameters>
    <asp:Parameter Name="old_id" Type="Int32" />
</DeleteParameters>
<UpdateParameters>
    <asp:Parameter Name="name" Type="String" />
    <asp:Parameter Name="is_group" Type="Boolean" />
    <asp:Parameter Name="old_id" Type="Int32" />
</UpdateParameters>
```

Необходимо минимально настроить и GridView:

```
<asp:GridView ID="gv" runat="server" DataSourceID="SqlDS"
                AutoGenerateColumns="False" DataKeyNames="id">
    <Columns>
        <asp:BoundField DataField="id" HeaderText="id"
                        InsertVisible="False" ReadOnly="True" />
        <asp:BoundField DataField="name" HeaderText="name" />
        <asp:CheckBoxField DataField="is_group" />
        <asp:CommandField ButtonType="Button"
                          ShowDeleteButton="True"
                          ShowEditButton="True" />
```

```
    </Columns>
</asp:GridView>
```

Кроме возможности декларативного взаимодействия с визуальными элементами управления, SqlDataSource предоставляет программную возможность вызова команд с помощью соответствующих методов – Select(), Insert(), Update() и Delete(). Метод Select() вызывается с параметром типа DataSourceSelectArguments и возвращает DataSet или IDataReader в зависимости от значения свойства DataSourceMode, остальные же методы вызываются без параметров и возвращают количество обработанных строк.

Рассмотрим пример использования методов SqlDataSource. В предыдущем примере не использовалось свойство InsertCommand, так как элемент управления GridView не умеет добавлять данные. Исправим это упущение с помощью дополнительных элементов управления - разместим на форме поле для ввода, переключатель и кнопку для добавления введенных данных в базу:

```
<p />Add Performer<p />
Name: <asp:TextBox ID="txtName" runat="server" /> <br />
Is Group: <asp:CheckBox ID="cbxGroup" runat="server" /> <br />
<asp:Button ID="btnAdd" runat="server" Text="Add" />
```

Добавим в описание SqlDataSource команду для вставки записи:

```
InsertCommand="INSERT INTO [Performers] ([name], [is_group])
               VALUES (@name, @is_group)"
. . .
<InsertParameters>
    <asp:ControlParameter ControlID="txtName" Name="name"
                          PropertyName="Text" Type="String" />
    <asp:ControlParameter ControlID="cbxGroup" Name="is_group"
                          PropertyName="Checked" Type="Boolean" />
</InsertParameters>
```

Далее в обработчике события Click кнопки нужно записать:

```
SqlDS.Insert();
```

При нажатии на кнопку Add в базу добавится новая запись, содержащая введенные значения. Аналогичным образом можно вызвать и остальные методы изменения данных.

SqlDataSource для каждой своей команды предоставляет пару событий вида *ИмяКоманды*ing и *ИмяКоманды*ed, которые происходят соответственно перед вызовом команды и сразу же после него. Кроме этого, SqlDataSource имеет событие Filtering, происходящее перед применением фильтра к полученным данным.

Элемент управления ObjectDataSource

Элемент управления ObjectDataSource работает только с бизнесс-объектами. При этом ObjectDataSource, аналогично SqlDataSource, может получать набор данных и производить изменения данных. Работа с элементом управления ObjectDataSource практически идентична работе с SqlDataSource с некоторыми небольшими изменениями.

Для кода примеров опишем класс Performer, который содержит методы для работы с таблицей исполнителей в базе данных.

```
public class Performer
{
    private const string connStr = "...";

    private const string INSERT_CMD =
        "INSERT INTO [Performers] ([name], [is_group])" +
        "VALUES (@name, @is_group)";

    private const string SELECT_CMD = "SELECT * FROM [Performers]";

    private const string UPDATE_CMD =
        "UPDATE [Performers] SET [name] = @name," +
        "[is_group] = @is_group WHERE [id] = @old_id";

    private const string DELETE_CMD =
        "DELETE FROM [Performers] WHERE [id] = @old_id";

    public void Create(string name, bool is_group)
    {
        var conn = new SqlConnection(connStr);
        var cmd = new SqlCommand(INSERT_CMD, conn);
        var name_param = new SqlParameter
        {
            ParameterName = "@name",
            SqlDbType = SqlDbType.NVarChar,
            Value = name
```

```
        };
        var isgroup_param = new SqlParameter
        {
            ParameterName = "@is_group",
            SqlDbType = SqlDbType.Bit,
            Value = is_group ? 1 : 0
        };
        cmd.Parameters.Add(name_param);
        cmd.Parameters.Add(isgroup_param);
        conn.Open();
        cmd.ExecuteNonQuery();
        conn.Close();
    }

    public SqlDataReader Read()
    {
        var conn = new SqlConnection(connStr);
        var cmd = new SqlCommand(SELECT_CMD, conn);
        conn.Open();
        return cmd.ExecuteReader(CommandBehavior.CloseConnection);
    }

    public void Update(int ID, string name, bool is_group)
    {
        var conn = new SqlConnection(connStr);
        var cmd = new SqlCommand(UPDATE_CMD, conn);
        var name_param = new SqlParameter
        {
            ParameterName = "@name",
            SqlDbType = SqlDbType.NVarChar,
            Value = name
        };
        var isgroup_param = new SqlParameter
        {
```

```
            ParameterName = "@is_group",
            SqlDbType = SqlDbType.Bit,
            Value = is_group ? 1 : 0
        };
        var id_param = new SqlParameter
        {
            ParameterName = "@old_id",
            SqlDbType = SqlDbType.Int,
            Value = ID
        };
        cmd.Parameters.Add(name_param);
        cmd.Parameters.Add(isgroup_param);
        cmd.Parameters.Add(id_param);
        conn.Open();
        cmd.ExecuteNonQuery();
        conn.Close();
    }

    public void Delete(int ID)
    {
        var conn = new SqlConnection(connStr);
        var cmd = new SqlCommand(DELETE_CMD, conn);
        var id_param = new SqlParameter
        {
            ParameterName = "@old_id",
            SqlDbType = SqlDbType.Int,
            Value = ID
        };
        cmd.Parameters.Add(id_param);
        conn.Open();
        cmd.ExecuteNonQuery();
        conn.Close();
    }
}
```

Метод для получения данных – в нашем случае это `Performer.Read()` – должен возвращать экземпляр класса, реализующего `IEnumerable`. Есть определенная зависимость других свойств `ObjectDataSource` от типа возвращаемого этим методом значения – как и для `SqlDataSource` кэшироваться могут только данные в `DataSet`, а `DataReader` не может делать пейджинг. При написании остальных методов нужно помнить о свойствах `ConflictDetection` и `OldValuesParameterFormatString`, имеющих то же значение, что и у `SqlDataSource`.

При наличии готового класса можно настроить элемент управления `ObjectDataSource` для работы с данными. Основные отличия от `SqlDataSource`: свойство `TypeName` должно содержать строку с именем класса бизнес логики, свойства работы с данными имеют окончание *`Method`* вместо используемого в `SqlDataSource` окончания *`Command`*.

```
<asp:ObjectDataSource ID="ObjectDS" runat="server"
    TypeName="Performer" SelectMethod="Read"
    DeleteMethod="Delete" UpdateMethod="Update"
    InsertMethod="Create"
    ConflictDetection="OverwriteChanges"
    OldValuesParameterFormatString="old_{0}"
    <InsertParameters>
        <asp:ControlParameter ControlID="txtName" Name="name"
                              PropertyName="Text" Type="String" />
        <asp:ControlParameter ControlID="cbxGroup" Name="is_group"
                             PropertyName="Checked" Type="Boolean" />
    </InsertParameters>
    <DeleteParameters>
        <asp:Parameter Name="id" Type="Int32" />
    </DeleteParameters>
    <UpdateParameters>
        <asp:Parameter Name="name" Type="String" />
        <asp:Parameter Name="is_group" Type="Boolean" />
        <asp:Parameter Name="id" Type="Int32" />
    </UpdateParameters>
</asp:ObjectDataSource>
```

Зачастую для каждой сущности базы данных существует класс для ее хранения и класс для манипуляций этой сущностью. В этом случае нужно указать в свойстве `DataObjectTypeName` имя класса сущности, а методы `Insert()`, `Delete()` и `Update()` класса бизнес логики должны принимать только один параметр указанного в

DataObjectTypeName типа. В случае использования значения CompareAllValues в свойстве ConflictDetection метод Update() должен принимать два параметра указанного в DataObjectTypeName типа – со старыми и новыми значениями.

Список событий элемента управления ObjectDataSource практически полностью совпадает с подобным списком элемента управления SqlDataSource. Кроме того, ObjectDataSource имеет три события, отвечающие за происходящее с экземпляром класса бизнес логики – ObjectCreating, ObjectCreated и ObjectDisposing. Событие ObjectCreating происходит перед созданием экземпляра класса и в обработчике этого события можно, например, создавать экземпляр класса бизнес логики в случае, если этот класс должен создаваться с использованием конструктора с параметрами.

5.10. ПРОВЕРОЧНЫЕ ЭЛЕМЕНТЫ УПРАВЛЕНИЯ

Кроме визуальных элементов управления, ASP.NET предоставляет набор проверочных элементов. *Проверочные элементы управления* налагают определенные разработчиком ограничения на данные, вводимые пользователем в формы. При настройке проверочный элемент связывается с элементом управления на форме. В качестве проверяемых могут выступать элементы HTMLInputText, HTMLTextArea, HTMLSelect, HTMLInputFile, TextBox, DropDownList, ListBox, RadioButtonList. В случае если проверка закончилась неудачей, проверочный элемент способен отобразить текстовое разъясняющее сообщение около проверяемого элемента.

В веб-приложении проверка данных, вводимым пользователем, может выполняться на стороне клиента, на стороне сервера или в обоих местах. Проверка на стороне клиента уменьшает количество обменов между клиентом и сервером, необходимых для успешного завершения формы. Однако клиентская проверка может быть выполнена не всегда. Во-первых, для выполнения проверки браузер должен поддерживать язык сценариев. Во-вторых, клиент часто не обладает достаточной информацией, требуемой для завершения проверки. Поэтому проверки на стороне клиента обычно используются в сочетании с проверками на стороне сервера. Достоинство проверочных элементов ASP.NET заключается в том, что они способны автоматически распознавать поддержку клиентом языка сценариев и в зависимости от этого генерировать клиентский либо серверный код проверки.

Рассмотрим общую архитектуру проверочных элементов. Любой проверочный элемент реализует интерфейс System.Web.UI.IValidator, который объявлен следующим образом:

```
interface IValidator
{
    string ErrorMessage { set; get; }
    bool IsValid { set; get; }
    void Validate();
}
```

Метод Validate() выполняет процедуру проверки, свойство IsValid указывает, успешно ли выполнилась проверка, а свойство ErrorMessage позволяет определить строку-сообщение в случае провала проверки.

Для всех проверочных элементов базовым является абстрактный класс BaseValidator (из пространства имен System.Web.UI.WebControls), основные элементы которого перечислены в табл. 23.

Таблица 23

Элементы класса `BaseValidator`

Имя элемента	Описание
`ControlToValidate`	Строка-идентификатор проверяемого элемента управления
`Display`	Свойство определяет, должно ли значение проверочного элемента занять некоторое пространство, если оно не выводится. Значение свойства – элемент перечисления `ValidatorDisplay`
`EnableClientScript`	Булево свойство, управляет использованием клиентского скрипта для проверки
`Enabled`	Булево свойство для включения или выключения проверочного элемента
`ErrorMessage`	Свойство-строка, позволяет установить или прочитать текстовое сообщение, которое отображается в элементе `ValidationSummary` при неуспешной проверке
`ForeColor`	Цвет строки проверочного элемента (по умолчанию – красный)
`IsValid`	Булево свойство, которое показывает, успешно ли выполнилась проверка
`SetFocusOnError`	Указывает, должен ли элемент управления, который не прошел проверку, получать фокус ввода ★
`Text`	Строка, которую отображает проверочный элемент при провале проверки
`ValidationGroup`	Имя проверочной группы, к которой принадлежит элемент управления ★
`Validate()`	Метод выполняет процедуру проверку и обновляет значение свойства `IsValid`

Класс `Page` хранит список всех проверочных элементов на странице в коллекции `Validators`. Класс `Page` также предоставляет метод `Validate()`, который применяется для коллективного вызова одноимённого метода всех проверочных элементов страницы. Этот метод вызывается на стороне сервера автоматически после загрузки состояния элементов управления страницы. Метод `Page.Validate()` устанавливает булево свойство страницы `IsValid`. Как правило, значение данного свойства проверяется в обработчике события `Page_Load`.

Опишем подробнее конкретные проверочные элементы ASP.NET.

Элемент: `RequiredFieldValidator`

Назначение: Используется для проверки того, что элемент управления не пуст или значение в нем изменено.

Специфичные свойства:

- `InitialValue` – проверка считается не пройденной, если при потере фокуса элементом управления значение в нем равно строке `InitialValue`. По умолчанию значение свойства – пустая строка.

Элемент: `CompareValidator`

Назначение: Применяется для сравнения двух полей формы или поля и константы. Может использоваться для того, чтобы проверить, соответствует ли значение поля определенному типу.

Специфичные свойства:

- `ControlToCompare` – строка, идентификатор того элемента управления, с которым сравнивается указанный в `ControlToValidate` элемент.
- `ValueToCompare` – значение (в виде строки), с которым сравнивается элемент, связанный с `CompareValidator`[1].
- `Operator` – операция сравнения. Тип свойства – перечисление `ValidationCompareOperator`, в которое входят следующие элементы: `Equal`, `NotEqual`,

[1] Не устанавливайте свойства `ControlToCompare` и `ValueToCompare` одновременно. В противном случае, преимущество имеет свойство `ControlToCompare`.

GreaterThan, GreaterThanEqual, LessThan, LessThanEqual, DataTypeCheck. Если Operator равен DataTypeCheck, то выполняется **только** проверка того, соответствует ли значение в элементе управления типу, заданному в свойстве Type.

- Type – тип, в который будет преобразовано значение в элементе управления перед проверкой. Свойство принимает значения из перечисления ValidationDataType с элементами String, Integer, Double, Date, Currency.

Элемент: RangeValidator

Назначение: Проверяет, входит ли значение элемента управления в указанный текстовый или числовой диапазон.

Специфичные свойства:

- MaximumValue, MinimumValue – строки, задающие диапазон проверки.
- Type – тип, в который будет преобразовано значение в элементе управления перед проверкой. Аналог соответствующего свойства из CompareValidator.

Элемент: `RegularExpressionValidator`

Назначение: Проверяет, удовлетворяет ли значение элемента управления заданному регулярному выражению.

Специфичные свойства:

- `ValidationExpression` – строка с регулярным выражением.

Элемент: `CustomValidator`

Назначение: Выполняет определенную пользователем проверку при помощи заданной функции (на стороне клиента, на стороне сервера или в обоих местах).

Специфичные свойства и события:

- `OnServerValidate` – обработчик этого события должен быть задан для проведения проверки на стороне сервера.
- `ClientValidationFunction` – строки с именем клиентской функции, которая будет использоваться для проверки. Так как проверка выполняется на клиенте, то проверочная функция должна быть включена в клиентский скрипт и может быть написана на JavaScript или на VBScript.

Элемент: `ValidationSummary`

Назначение: Элемент может использоваться для отображения на странице итогов проверок. Если у проверочных элементов определены свойства `ErrorMessage`, то элемент покажет их в виде списка.

Специфичные свойства:

- `DisplayMode` – свойство позволяет выбрать вид суммарного отчета об ошибках. Значения свойства – элемент перечисления `ValidationSummaryDisplayMode`: `BulletList` (по умолчанию), `List`, `SingleParagraph`.
- `HeaderText` – строка с заголовком отчета.
- `ShowMessageBox` – если это булево свойство установлено в `true`, то отчет отображается в отдельном диалоговом окне.
- `ShowSummary` – если свойство установлено в `true` (по умолчанию), то отчет отображается на странице. Это свойство часто используется совместно с `ShowMessageBox`.

Проверочные группы

В ASP.NET 2.0 введено новое свойство - `ValidationGroup`. Оно имеется у проверочных элементов, элементов управления `input`-типа и кнопок. Пользоваться этим свойством просто: достаточно задать один и тот же идентификатор группы для выбранных проверочных элементов, и этот же идентификатор присвоить свойству `ValidationGroup` кнопки, с помощью которой будет выполняться активизация указанных элементов. Вот пример кода:

```
<asp:TextBox ID="tbx1" runat="server" />
<asp:RequiredFieldValidator ID="reqValidator1" runat="server"
                  ValidationGroup="Group1" ControlToValidate="tbx1"
                  ErrorMessage="tbx1 is mandatory" />

<asp:TextBox ID="tbx2" runat="server" />
<asp:RequiredFieldValidator ID="reqValidator2" runat="server"
                  ValidationGroup="Group2" ControlToValidate="tbx2"
                  ErrorMessage="tbx2 is mandatory" />

<asp:Button ID="btn1" runat="server" Text="Check Group1"
```

```
            ValidationGroup="Group1" />
<asp:Button ID="btn2" runat="server" Text="Check Group2"
            ValidationGroup="Group2" />
```

Здесь два элемента управления `RequiredFieldValidator` принадлежат к разным проверочным группам – `Group1` и `Group2`. Первая кнопка служит для проверки элементов управления из группы `Group1`, а вторая - для проверки элементов из группы `Group2`. Действуя таким же образом, процесс проверки можно сделать сколь угодно гранулярным.

Поддержка AJAX в ASP.NET MVC

Для поддержки технологии AJAX в приложениях ASP.NET MVC можно воспользоваться методами расширения класса `System.Web.Mvc.AjaxHelper`: `ActionLink()`, `RouteLink()`, `BeginForm()` и `BeginRouteForm()`. В классе `ViewPage` определено свойство `Ajax`, имеющее тип `AjaxHelper`. Так как работа указанных методов расширения базируется на клиентских JavaScript-библиотеках `MicrosoftAjax.js` и `MicrosoftMvcAjax.js`, необходимо подключить эти библиотеки в представлении или на эталонной странице.

Рассмотрим применение метода `Ajax.ActionLink()`. Создадим представление для вывода времени в различных часовых зонах. При выборе зоны запрос будет отсылаться серверу асинхронно.

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

<!doctype html public "-//w3c//dtd xhtml 1.0 strict//en"
               "http://www.w3.org/tr/xhtml1/dtd/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <script src="<%= Url.Content("~/Scripts/MicrosoftAjax.js") %>"
            type="text/javascript"></script>
    <script src="<%= Url.Content("~/Scripts/MicrosoftMvcAjax.js")%>"
            type="text/javascript"></script>
</head>
<body>
    <h2>Который час?</h2>
    <p>Показать время в зоне:
        <%= Ajax.ActionLink("UTC", "GetTime", new { zone = "utc" },
                new AjaxOptions { UpdateTargetId = "myRes" }) %>
        <%= Ajax.ActionLink("BST", "GetTime", new { zone = "bst" },
                new AjaxOptions { UpdateTargetId = "myRes" }) %>
        <%= Ajax.ActionLink("MDT", "GetTime", new { zone = "mdt" },
```

```
                new AjaxOptions { UpdateTargetId = "myRes" }) %></p>
    <div id="myRes" style="border: 2px dotted red; padding: .5em;">
        Результат здесь
    </div>
    <p>Страница сгенерирована
        <%= DateTime.UtcNow.ToString("h:MM:ss tt") %>(UTC)</p>
</body>
</html>
```

Обратите внимание на теги, которые подключают клиентские JavaScript-библиотеки. Метода `Ajax.ActionLink()` в нашем примере принимает как параметр текст ссылки, имя действия (`"GetTime"`), набор параметров для действия (`new { zone = "utc" }`) и указание на то, какой HTML-элемент следует обновить при получении ответа от сервера (`UpdateTargetId = "myRes"`).

Чтобы сделать пример полностью работоспособным, требуется в контроллере описать действие, вызываемое при AJAX-запросе. Для нашего случая достаточно, чтобы действие генерировало результат в виде простой строки.

```
public string GetTime(string zone)
{
    DateTime time = DateTime.UtcNow.AddHours(offsets[zone]);
    return string.Format(
                "<div>Время в зоне {0} : {1:h:MM:ss tt}</div>",
                zone.ToUpper(), time);
}

private Dictionary<string, int> offsets =
                    new Dictionary<string, int> { { "utc", 0 },
                                { "bst", 1 }, { "mdt", -6 }};
```

Заметим, что если браузер клиента не поддерживает JavaScript, метод `Ajax.ActionLink()` работает как обычная ссылка, генерируемая `Html.ActionLink()`. В серверном методе можно распознать AJAX-запросы при помощи булевской функции `Request.IsAjaxRequest()`.

```
public ActionResult GetTime(string zone)
{
    DateTime time = DateTime.UtcNow.AddHours(offsets[zone]);
    if (Request.IsAjaxRequest())
    {
```

```
            return Content(string.Format(
                        "<div>Время в зоне {0} : {1:h:MM:ss tt}</div>",
                        zone.ToUpper(), time));
        }
        else
        {
            // TODO: необходимо описать представление GetTime
            return View(time);
        }
    }
```

Метод `Ajax.ActionLink()` имеет множество перегруженных версий, которые в отношении параметров в основном соответствуют версиям метода `Html.ActionLink()`. Однако `Ajax.ActionLink()` принимает дополнительный параметр типа `AjaxOptions` для конфигурирования AJAX-запроса. Параметры конфигурации представлены в табл. 5. Все параметры являются необязательными, кроме `UpdateTargetId`.

Таблица 5

Свойства класса `AjaxOptions`

Имя свойства	Описание
`Confirm`	Если задано это строковое свойство, перед отправкой AJAX-запроса показывается окно для подтверждения или отмены запроса
`HttpMethod`	Имя HTTP-метода (глагола), используемого при отправке запроса
`InsertionMode`	Режим работы с содержимым HTML-элемента для приёма ответа сервера (замещение содержимого, дополнение)
`LoadingElementId`	Идентификатор HTML-элемента, который становится видимым при отправке запроса и скрывается при получении ответа от сервера. Обычно свойство используется для *индикаторов запроса*
`OnBegin`	Имя JS-функции, вызываемой перед началом AJAX-запроса. Запрос не выполняется, если функция возвращает значение `false`
`OnComplete`	Имя JS-функции, вызываемой после завершения AJAX-запроса. Результат запроса игнорируется, если функция возвращает `false`
`OnSuccess`	Имя JS-функции, вызываемой после успешного завершения AJAX-запроса (вызывается после функции, указанной в `OnComplete`)
`OnFailure`	Имя JS-функции, вызываемой после неудачного завершения AJAX-запроса (вызывается после функции, указанной в `OnComplete`)
`UpdateTargetId`	Идентификатор того HTML-элемента, в который будет помещён ответ сервера
`Url`	Если значение указано, AJAX-запрос направляется на этот адрес

Иногда в AJAX-запрос необходимо включить пользовательские данные. В этом случае следует использовать метод `Ajax.BeginForm()`. Его параметры аналогичны

параметрам метода `Html.BeginForm()` с дополнительным параметром типа `AjaxOptions`.

Ajax-Формы

Итак, после настройки и подключения всех необходимых скриптов мы можем приступить непосредственно к работе с Ajax. Допустим, у нас есть класс `Book`, содержащий данные о книге, а в БД у нас может находиться несколько книг одного автора. И нам надо реализовать поиск всех книг определенного автора.

Казалось бы, зачем в данном случае Ajax, если мы можем, например, в форму вводить имя автора и отправлять на сервер, а сервер в качестве ответа возвратит нам страницу с нужным результатом. Но, как выше уже говорилось, AJAX поможет нам избежать перезагрузки всей страницы и выполнить загрузку данных в **асинхронном режиме**, что несомненно повышает производительность приложения.

Для начала определим действие контроллера, которое будет отвечать за извлечение из БД нужной информации и передавать извлеченную информацию в частичное представление:

```
public ActionResult BookSearch(string name)
{
    var allbooks = db.Books.Where(a => a.Author.Contains(name));
    return PartialView(allbooks);
}
```

Итак, действие получает в качестве параметра имя автора и по нему осуществляет поиск в БД. Теперь добавим к представлениям данного контроллера частичное представление BookSearch.cshtml. поскольку частичные представления довольно удобны для работы с AJAX:

```
@model IEnumerable<AjaxMvcApplication.Models.Book>

<div id="searchresults">
    @if (Model != null && Model.Count()>0)
    {
        <h3>Все книги автора : @Model.First().Author</h3>
        <ul>
        @foreach (var item in Model)
        {
            <li>@item.Name</li>
```

```
        }
        </ul>
    }
</div>
```

В данном случае представление типизируется для модели `IEnumerable<AjaxMvcApplication.Models.Book>`, которая и будет передаваться в представление. А затем в элемент `div` будут выводиться результаты поиска в виде списка при условии, конечно, если модель не равна `null`.

Теперь перейдем к самому представлению, которое и будет отображаться пользователю:

```
@{
    ViewBag.Title = "Index";
}

<div>
@using (Ajax.BeginForm("BookSearch", new AjaxOptions { UpdateTargetId = "searchresults"}))
{
    <input type="text" name="name" />
    <input type="submit" value="Поиск" />
}
@Html.Partial("BookSearch")
</div>
```

Хелпер **Ajax.BeginForm** похож на хелпер `Html.BeginForm` - он также создает элемент `form`, который используется для отправки запроса на сервер. Первый параметр принимает имя действия, к которому будет обращен запрос. В данном случае это созданное выше действие `BookSearch`, которое возвращает частичное представление с данными. Если действие находится не в текущем контроллере, а в другом мы также можем указать имя контроллера: `Ajax.BeginForm("BookSearch", "Home", new AjaxOptions....`

Второй параметр более интересный. Он представляет объект **AjaxOptions**, который влияет на отображение результатов. Он принимает ряд параметров, из которых мы в данном случае использовали только `UpdateTargetId`. Этот параметр указывает, что у

нас при получении результатов от сервера на странице будет обновляться элемент с id `searchresults`. А это как раз элемент выше созданного частичного представления.

В конце бы выводим само частичное представление на страницу:
`@Html.Partial("BookSearch").`

Но нам необязательно выводить частичное представление. Мы могли бы обойтись обычным элементом `div`. Так выше определенный код представления по своему функционалу будет аналогичен следующему:

```
@{
    ViewBag.Title = "Index";
}

<div>
@using (Ajax.BeginForm("BookSearch", new AjaxOptions { UpdateTargetId = "results"}))
{
    <input type="text" name="name" />
    <input type="submit" value="Поиск" />
}
<div id="results"></div>
</div>
```

В данном случае опять же мы выводим результаты запроса в элемент с id, указанным в параметре `UpdateTargetId`. Теперь мы можем запустить приложение и осуществить поиск с помощью AJAX:

Обратите внимание на создаваемую разметку для данной формы:

```
<form action="/Home/BookSearch" data-ajax="true" data-ajax-mode="replace" data-ajax-update="#results" id="form0" method="post">
    <input type="text" name="name" />
    <input type="submit" value="Поиск" />
</form>
<div id="results"></div>
```

Создается как и в случае с хелпером `Html.BeginForm` элемент `form`. Но теперь в специальном атрибуте указывается, что это Ajax-форма: `data-ajax="true"`. Остальные параметры формы являются передачей в html параметров объекта AjaxOptions.