Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра информатики

А.А. Волосевич

ЯЗЫК C# И ПЛАТФОРМА .NET
(часть 5)

Курс лекций
для студентов специальности I-31 03 04 «Информатика»
всех форм обучения

Минск 2009

СОДЕРЖАНИЕ

5. ТЕХНОЛОГИЯ ASP.NET 4

5.1. ОБЩИЕ КОНЦЕПЦИИ ASP.NET 4

5.2. ВЕБ-ПРИЛОЖЕНИЕ 7

5.3. СТРУКТУРА ASPX-ФАЙЛА 8

5.4. ДИРЕКТИВЫ СТРАНИЦЫ 10

5.5. КЛАСС SYSTEM.WEB.UI.PAGE 14

5.6. ЖИЗНЕННЫЙ ЦИКЛ СТРАНИЦЫ 19

5.7. ОБЩИЙ ОБЗОР СЕРВЕРНЫХ ЭЛЕМЕНТОВ УПРАВЛЕНИЯ 26

5.8. ЭЛЕМЕНТЫ УПРАВЛЕНИЯ HTML 30

5.9. ЭЛЕМЕНТЫ УПРАВЛЕНИЯ WEB 34

5.10. ПРОВЕРОЧНЫЕ ЭЛЕМЕНТЫ УПРАВЛЕНИЯ 39

5.11. ЭТАЛОННЫЕ СТРАНИЦЫ И ТЕМЫ 43

5.12. НЕКОТОРЫЕ ПРИЁМЫ РАБОТЫ СО СТРАНИЦАМИ 49

5.13. СВЯЗЫВАНИЕ С ДАННЫМИ 53

5.14. СПИСКОВЫЕ ЭЛЕМЕНТЫ УПРАВЛЕНИЯ 55

5.15. ЭЛЕМЕНТЫ УПРАВЛЕНИЯ ДЛЯ ИСТОЧНИКОВ ДАННЫХ 57

5.16. ТАБЛИЦЫ, СВЯЗАННЫЕ С ДАННЫМИ 65

5.17. ОТОБРАЖЕНИЕ ОТДЕЛЬНЫХ ЗАПИСЕЙ 71

5.18. КОНФИГУРИРОВАНИЕ ВЕБ-ПРИЛОЖЕНИЙ 74

5.19. ИНФРАСТРУКТУРА ОБРАБОТКИ ЗАПРОСА 75

5.20. ВЕБ-ПРИЛОЖЕНИЕ И ФАЙЛ GLOBAL.ASAX 79

5.21. МОДЕЛЬ ПОСТАВЩИКОВ 81

5.22. ПОДДЕРЖКА СОХРАНЕНИЯ СОСТОЯНИЯ 83

5.23. КЭШИРОВАНИЕ В ASP.NET 89

5.24. ОБЕСПЕЧЕНИЕ БЕЗОПАСНОСТИ В ВЕБ-ПРИЛОЖЕНИЯХ 94

5.25. УПРАВЛЕНИЕ ЧЛЕНСТВОМ И РОЛЯМИ 98

5.26. ПРОФИЛИ ПОЛЬЗОВАТЕЛЯ 106

5.27. ЛОКАЛИЗАЦИЯ И РЕСУРСЫ 109

5.28. ПОЛЬЗОВАТЕЛЬСКИЕ ЭЛЕМЕНТЫ УПРАВЛЕНИЯ 111

5. ТЕХНОЛОГИЯ ASP.NET

5.1. ОБЩИЕ КОНЦЕПЦИИ ASP.NET

В этом параграфе рассматриваются базовые концепции и понятия технологии ASP.NET. 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. В настоящее время подобный подход не соответствует большинству информационных услуг, предоставляемых с помощью сети Интернет[1]. 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.

[1] Классическим примером является типичный Интернет-магазин. Если пользователь на странице поиска определяет условие фильтрации, то браузер должен отобразить список товаров, удовлетворяющих фильтру. Естественно, этот процесс подразумевает динамическую генерацию страницы с результатами на стороне сервера.

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 пользова-

тельских 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*. В ASP.NET 1.0 динамически компилировались только файлы Code Behind. Если же приложение использовало сторонние классы, то сборки с данными классами должны были быть скомпилированы заранее и размещаться либо в GAC, либо в специальном подкаталоге веб-приложения. 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*. Страницы практически любого сайта имеют единообразную структуру и однотипное оформление. 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. Сюда относятся свойства элементов управления, таблицы стилей страницы, изображения и шаблоны страниц. 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 новая, так называемая 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.

В платформе .NET версии 3.5 технология ASP.NET обогатилась новыми элементами управления, а также полноценной поддержкой программирования с использованием AJAX.

5.2. ВЕБ-ПРИЛОЖЕНИЕ

Под веб-приложением будем понимать совокупность файлов и каталогов, размещенных в отдельном каталоге, которому сопоставлен виртуальный каталог веб-сервера. Рабочий процесс ASP.NET обслуживает каждое веб-приложение в отдельном домене, используя специфичные настройки приложения. *Границей* приложения является виртуальный каталог, в том смысле, что, перемещаясь по страницам внутри виртуального каталога, пользователь остаётся в рамках одного веб-приложения.

Любое веб-приложение может содержать следующие элементы:

1. *Страницы ASP.NET*: набор файлов с расширением `.aspx` - *файлы разметки страницы* и, возможно, парные им файлы кода на каком-либо языке программирования (C#, VB.NET) – *файлы Code Behind*. Страницы ASP.NET могут размещаться как в корне веб-приложения, так и в подкаталогах.

2. Один или несколько *файлов конфигурации* `web.config`. Если веб-приложение содержит подкаталоги, то допускается не более одного файла `web.config` на подкаталог.

3. *Пользовательские элементы управления* – файлы с расширением `.ascx` и, возможно, парные им файлы Code Behind.

4. Некоторые из перечисленных в табл. 1 специальных подкаталогов.

Таблица 1

Специальные подкаталоги веб-приложения

Имя	Что содержит
`Bin`	Скомпилированные сборки, необходимые для работы приложения
`App_Browsers`	Файлы с информацией о возможностях браузеров
`App_Code`	Файлы исходного кода классов (`.cs` или `.vb`), которые будут использоваться страницами
`App_Data`	Файлы данных приложения. Это могут быть XML-файлы или файлы баз данных
`App_GlobalResources`	Глобальные для приложения файлы ресурсов (`.resx`)
`App_LocalResources`	Файлы ресурсов (`.resx`) для отдельных страниц
`App_Themes`	Определения поддерживаемых приложением тем
`App_WebReferences`	Файлы `.wsdl`, необходимые для связывания веб-сервисов с приложением

5. *Файлы эталонных страниц* – файлы с расширением `.master` и, возможно, парные им файлы Code Behind.

6. Единственный файл `global.asax`, размещаемый в корневом каталоге приложения. Этот файл играет роль пункта реализации глобальных (для приложения) событий, объектов и переменных

7. Файлы любых других типов (`*.html`, `*.xml`, изображения и т. д.), возможно, размещенные в отдельных подкаталогах.

Будем рассматривать интегрированную среду Visual Studio 2008 как основное средство для создания веб-приложений. Использование VS 2008 обеспечивает следующие удобства:

- Автоматическое создание некоторых необходимых элементов веб-приложения (подкаталогов) при выборе соответствующего типа проекта.
- Возможность визуального редактирования страниц.
- Возможности подсветки синтаксиса и IntelliSense.
- Встроенный веб-сервер для запуска и отладки приложений.

5.3. СТРУКТУРА ASPX-ФАЙЛА

Задачей параграфа является указание элементов, образующих файл разметки страницы `.aspx`. Напомним, что наряду с файлом разметки страницы, составной частью страницы является, как правило, и файл Code Behind.

Рассмотрим пример простой aspx-страницы.

```
<%-- 1. Директива страницы --%>
<%@ Page Language="C#" AutoEventWireup="true"
         CodeBehind="Default.aspx.cs" Inherits="WebApp._Default" %>

<%-- 4. Блок серверного кода --%>
<script runat="server">
    protected void btnSend_Click(object sender, EventArgs e)
    {
        tbxName.Text = tbxName.Text.ToUpper();
    }
</script>

<%-- 3. Обычный HTML --%>
<html>
<body>
    <%-- 5. Блок рендерного кода --%>
    <% for (int i = 0; i < 10; i++) Response.Write("Hello"); %>
    <form id="form1" runat="server">
    <div>
        Input name:
        <%-- 2. Серверные элементы управления --%>
        <asp:TextBox ID="tbxName" runat="server" />
        <asp:Button ID="btnSend" runat="server"
                    OnClick="btnSend_Click" />
```

```
    </div>
    </form>
</body>
</html>
```

Выделим следующие элементы страницы.

1. Директивы страницы.

Директивы используются для установки отдельных параметров страницы, таких как язык программирования для кода страницы или подключение пространства имен. Директива начинается с символа `@`, за которым следует имя директивы и набор атрибутов. Директивы могут размещаться в любом месте страницы, но вне HTML-элементов. Как правило, директивы помещают в начале страницы.

2. Серверные элементы управления и серверная форма.

Серверные элементы управления описываются с помощью специальных тэгов с обязательным атрибутом `runat="server"`. Они соответствуют полям в классе страницы. Серверный элемент управления обладает набором свойств, установка которых возможна на странице как задание соответствующий атрибутов. Все серверные элементы должны быть размещены в пределах *серверной формы* (`<form runat="server">`).

3. HTML-код. HTML-код не обрабатывается процессом ASP.NET специальным образом, а сразу пересылается клиенту. Выводом HTML-кода занимается специальный внутренний метод класса, который соответствует странице.

4. Блоки серверного кода. Это блоки, обрамленные тэгом `<script>` с обязательным атрибутом `runat="server"`. Блоки серверного кода транслируются в члены класса, соответствующего странице. В приведенном примере класс будет содержать метод `btnSend_Click()`. Отметим, что наличие блоков серверного кода на странице противоречит концепции Code Behind. Согласно данной концепции код, связанный со страницей, должен быть помещен в отдельный класс, от которого наследуется класс страницы.

5. Блоки рендерного кода. Блоки рендерного кода используются для генерации потока вывода. При обработке на сервере блоки рендерного кода помещаются непосредственно в метод, выполняющий вывод HTML-кода. Если блок рендерного кода записывается в форме `<%= выражение %>`, то в метод вставляется вывод вычисленного выражения.

Кроме упомянутых выше элементов, страница ASP.NET может содержать комментарии, блоки привязки данных (они будут подробно рассмотрены далее), клиентские скрипты.

5.4. ДИРЕКТИВЫ СТРАНИЦЫ

Директивы страницы служат для конфигурирования среды её выполнения. Список основных директив приведён в табл. 2[1]. Самой важной и часто используемой директивой является `@Page`.

Таблица 2. Директивы страницы

Директива	Описание
`@Assembly`	Связывает сборку с текущей страницей или пользовательским элементом управления
`@Control`	Позволяет задать атрибуты, специфические для пользовательских элементов управления и влияющие на их компиляцию
`@Implements`	Указывает, что страница или пользовательский элемент управления реализует заданный интерфейс
`@Import`	Задает пространство имен, которое должно быть импортировано в страницу или пользовательский элемент управления
`@Master`	Идентифицирует эталонную страницу ★[2]
`@OutputCache`	Определяет политики кэширования вывода страницы
`@Page`	Позволяет задать специфичные для страниц атрибуты, управляющие поведением синтаксического анализатора и компилятора
`@Reference`	Связывает страницу или пользовательский элемент управления с текущей страницей или пользовательским элементом управления
`@Register`	Определяет для страницы или элемента управления пользовательский тэг. Этот новый тэг (идентифицируемый префиксом и именем) связывается с пространством имен и кодом пользовательского элемента управления

Все директивы, за исключением `@Page`, `@Master` и `@Control`, могут использоваться как в файлах страниц, так и в файлах элементов управления. Директивы `@Page` и `@Control` являются взаимоисключающими: первая может использоваться только в файлах `.aspx` (страницах), а вторая - только в файлах `.ascx` (элементах управления). Что касается директивы `@Master`, то она идентифицирует страницу особого вида - эталонную страницу.

Каждая директива имеет собственный набор типизированных атрибутов. Присвоение атрибуту значения неверного типа или использование в директиве не поддерживаемого ею атрибута приводит к ошибке компиляции.

Любой файл `.aspx` может содержать директиву `@Page`, и хотя её наличие не является обязательным, на практике она требуется каждой более или менее сложной странице. У директивы `@Page` около тридцати атрибутов, которые могут быть логически разделены на три категории: связанные с компиляцией (табл. 3), общим поведением страницы (табл. 4) и её выводом (табл. 5). Страни-

[1] Имя директивы чувствительно к регистру.
[2] Символ ★ здесь и далее означает новый элемент, появившийся в ASP.NET 2.0.

ца ASP.NET компилируется при первом обращении к ней, и отправляемый браузеру HTML-код генерируется методами динамически создаваемого класса. Атрибуты, перечисленные в табл. 3, позволяют настраивать параметры компилятора.

Таблица 3

Атрибуты директивы `@Page`, связанные с компиляцией

Атрибут	Описание
`ClassName`	Имя класса (без пространства имен), который будет динамически компилироваться по запросу страницы
`CodeFile`	Путь к файлу отделенного кода текущей страницы. Файл исходного кода должен быть развернут на веб-сервере ★
`CodeBehind`	Путь к файлу отделенного кода текущей страницы, подлежащему компиляции в развертываемую сборку
`CodeFileBaseClass`	Родительский класс родительского класса страницы. Иными словами, в этом атрибуте задается имя класса, от которого должен быть производным класс отделенного кода страницы ★
`CompilationMode`	Одно из трех значений, указывающих, должна ли страница компилироваться во время выполнения: `Never`, `Auto` или **`Always`**[1] ★
`CompilerOptions`	Опции компилятора
`Debug`	Булево значение, указывающее, должна ли страница компилироваться с отладочными символами
`Explicit`	Булево значение, указывающее, должна ли страница, языком которой является VB.NET, компилироваться с установкой `Option Explicit On`
`Inherits`	Базовый класс, наследуемый страницей. Им может быть любой класс, производный от `System.Web.UI.Page`
`Language`	Язык, который должен использоваться при компиляции встроенных блоков кода. Поддерживаются языки **`VB.NET`**, `C#`, `JScript.NET` и `J#`
`MasterPageFile`	Эталонная страница для текущей страницы ★
`Src`	Файл исходного кода, содержащий реализацию базового класса, заданного в атрибуте `Inherits` (атрибут не используется Visual Studio)
`Strict`	Булево значение, указывающее, должна ли страница, языком которой является VB.NET, компилироваться с установкой `Option Strict On`
`Trace`	Булево значение, указывающее, включена ли трассировка страницы. Если трассировка включена, в вывод страницы добавляется трассировочная информация. По умолчанию – `false`
`TraceMode`	Указывает, как при включенной трассировке должны отображаться трассировочные сообщения. Допустимыми значениями являются **`SortByTime`** и `SortByCategory`
`WarningLevel`	Определяет степень важности предупреждения компилятора, достаточную для прекращения им компиляции страницы. Допустимы значения от 0 до 4

Атрибуты, перечисленные в табл. 4, обеспечивают контроль над общим поведением страницы и поддерживаемыми возможностями. Например, с их

[1] Полужирным шрифтом выделены значения атрибутов, используемые по умолчанию.

помощью можно задать пользовательскую страницу сообщения об ошибке, отключить поддержку состояния сеанса, управлять транзакционным поведением страницы.

Таблица 4

Атрибуты директивы `@Page`, определяющие поведение страницы

Атрибут	Описание
`AspCompat`	Если атрибут имеет значение `true`, разрешено выполнение страницы в однопоточном апартаментном режиме (STA)
`Async`	Если атрибут имеет значение `true`, генерируемый класс страницы реализует не `IHttpHandler`, а `IHttpAsyncHandler` (функция поддержки асинхронного выполнения страницы) ★
`AutoEventWireup`	Булев атрибут; указывает, должны ли события подключаться к соответствующим элементам управления автоматически (исходя из имени обработчика)
`Buffer`	Булево значение, определяющие, включена ли буферизация ответов HTTP. По умолчанию атрибут имеет значение `true`
`Description`	Текстовое описание страницы. Атрибут предназначен для целей документирования
`EnableSessionState`	Способ работы страницы с данными состояния сеанса. Если этот атрибут имеет значение `true`, состояние сеанса считывается и записывается, при значении `false` оно недоступно приложению, а при значении `ReadOnly` состояние сеанса может считываться, но не изменяться
`EnableViewState`	Булев атрибут, указывающий, сохраняется ли состояние представления страницы между ее запросами
`EnableViewStateMac`	Булев атрибут, указывающий, должна ли ASP.NET вычислять машинно-зависимый аутентификационный код и применять его к состоянию представления страницы (в дополнение к кодированию методом Base64)
`ErrorPage`	Определяет целевой URL, по которому пользователь будет автоматически перенаправлен, если при выполнении страницы произойдет необработанное исключение
`SmartNavigation`	Булев атрибут, указывающий, поддерживает ли страница функции интеллектуальной навигации, позволяющие обновлять страницу, не теряя позицию прокрутки и фокус элемента
`Theme`, `StylesheetTheme`	Имя темы (или темы таблицы стилей) для страницы ★
`Transaction`	Указывает, поддерживает ли страница транзакции. Допустимыми значениями являются: **`Disabled`**, `NotSupported`, `Supported`, `Required` и `RequiresNew`
`ValidateRequest`	Булев атрибут, указывающий, должна ли производиться проверка запроса. Если атрибут имеет значение **`true`**, ASP.NET сверяет все вводимые данные с жестко закодированным списком потенциально опасных значений, что позволяет уменьшить риск межсайтовых сценарных атак ★

Атрибуты, перечисленные в табл. 5, позволяют управлять форматом генерируемого страницей вывода. Например, вы можете задать тип контента страницы или локализовать вывод.

Таблица 5

Атрибуты директивы @Page, влияющие на вывод страницы

Атрибут	Описание
ClientTarget	Целевой браузер, для которого должен быть адаптирован выводимый серверными элементами управления контент
CodePage	Номер кодовой страницы для ответа. Атрибут следует устанавливать только в том случае, если создается страница с использованием кодовой страницы, отличной от заданной по умолчанию для веб-сервера
ContentType	Тип контента ответа (один из стандартных MIME-типов)
Culture	Культура страницы. От этой установки зависят такие параметры, как система записи и сортировки текста, календарь, форматы даты и времени. Имя культуры должно включать как идентификатор языка, так и идентификатор страны. Например, допустимым является идентификатор en-US
LCID	32-разрядный идентификатор культуры страницы. По умолчанию используется идентификатор культуры веб-сервера
ResponseEncoding	Идентификатор кодировки символов страницы. Это значение используется для установки атрибута CharSet в заголовке HTTP Content-Type
UICulture	Имя задаваемой по умолчанию культуры, используемое менеджером ресурсов для поиска культуро-зависимых ресурсов во время выполнения приложения

Директива @Assembly связывает с текущей страницей сборку, классы и интерфейсы которой должны быть доступны этой странице. Компилируя страницу, ASP.NET автоматически связывает с ней несколько сборок, определенных как используемые по умолчанию. Помимо этих сборок исполняющая среда ASP.NET автоматически связывает со страницей все те сборки, которые она находит в папке Bin приложения.

Для того чтобы связать сборку со страницей, включите в файл последней одну из двух следующих директив:

```
<%@ Assembly Name="имя_сборки" %>

<%@ Assembly Src="код_сборки.cs" %>
```

У директивы @Assembly есть два взаимоисключающих атрибута: Name и Src. Атрибут Name определяет имя сборки, которая должна быть связана со страницей (это имя не содержит пути и расширения). В атрибуте Src указывается путь к исходному файлу, подлежащему динамической компиляции и последующему связыванию со страницей.

Директива @Import используется для импорта пространств имен для страницы. Пример использования директивы:

```
<%@ Import Namespace="CommonNS" %>
```

Одна директива @Import позволяет импортировать одно пространство имен, поэтому для импортирования нескольких пространств надо повторить

директиву. При работе с ASP.NET автоматически импортирует следующие пространства имен:

```
System                              System.Web.Profile
System.Collections                  System.Web.Security
System.Collections.Specialized      System.Web.SessionState
System.Configuration                System.Web.UI
System.Text                         System.Web.UI.HtmlControls
System.Text.RegularExpressions      System.Web.UI.WebControls
System.Web                          System.Web.UI.WebControls.WebParts
System.Web.Caching
```

Директива `@Implements` позволяет реализовать на странице интерфейс. При реализации интерфейса вы сообщаете, что страница будет поддерживать определенные свойства, методы и события (аналогично реализации интерфейса в классе). В следующем примере указано, что страница реализует интерфейс `IPostBackEventHandler`:

```
<%@ Implements Interface="IPostBackEventHandler" %>
```

Директива `@Reference` устанавливает динамическую связь между текущей страницей и другой страницей или пользовательским элементом управления. Директива может встречаться в теле страницы несколько раз и имеет два взаимоисключающих атрибута - `Page` и `Control`, в каждом из которых задается путь к некоторому исходному файлу:

```
<%@ Reference Page="страница" %>

<%@ Reference Control="пользовательский_элемент_управления" %>
```

Атрибут `Page` указывает на исходный файл `.aspx`, то есть на файл страницы, а атрибут `Control` - на исходный файл `.ascx`, то есть на файл пользовательского элемента управления. В обоих случаях заданный исходный файл динамически компилируется в сборку и определенные в нем классы становятся программно доступными странице, содержащей директиву `@Reference`.

Директива `@Register` используется при добавлении на страницу индивидуального серверного элемента управления для сообщения компилятору информации об этом элементе. Директива `@OutputCache` управляет кэшем вывода для страницы или пользовательского элемента управления. Подробный синтаксис этих директив будет рассмотрен в соответствующих параграфах.

5.5. КЛАСС SYSTEM.WEB.UI.PAGE

Любая aspx-страница компилируется в объект определенного класса. Этот класс является наследником класса `System.Web.UI.Page`.

Рассмотрим свойства класса `Page` подробнее. Для удобства разделим их на несколько групп. В табл. 6 приведен список свойств, возвращающих внутрен-

ние объекты страницы. Эти объекты являются важными элементами инфраструктуры, обеспечивающей выполнение страницы.

Таблица 6

Внутренние объекты класса `System.Web.UI.Page`

Имя свойства	Описание
`Application`	Объект класса `HttpApplicationState`, описывающий состояние приложения, к которому относится страница. Для отдельного веб-приложения существует ровно один объект `Application`, который используется всеми клиентами
`Cache`	Объект класса `Cache`, ссылка на кэш данных веб-приложения
`Request`	Объект `HttpRequest` – информация о HTTP-запросе
`Response`	Объект `HttpResponse` – информация о HTTP-ответе
`Server`	Объект класса `HttpServerUtility`, описывающий параметры веб-сервера
`Session`	Объект класса `HttpSessionState`, хранящий данные текущей сессии пользователя в веб-приложении
`Trace`	Объект класса `TraceContext`. Если на странице разрешена трассировка, то можно пользоваться данным объектом для записи особой информации в журнал трассировки
`User`	Ссылка на объект, реализующий интерфейс `IPrincipal` и описывающий пользователя, от которого поступил запрос

В табл. 7 перечислены свойства `Page`, содержащие важную информацию и используемые при реализации определенной функциональности.

Таблица 7

Рабочие свойства класса `System.Web.UI.Page`

Имя свойства	Описание
`ClientScript`	Объект класса `ClientScriptManager`, содержащий клиентский сценарий, используемый в составе страницы ★
`Controls`	Коллекция дочерних элементов управления страницы
`ErrorPage`	Страница с сообщения об ошибке, куда в случае обнаружения необработанного исключения будет перенаправлен браузер
`Form`	Возвращает текущий объект `HtmlForm` для страницы ★
`Header`	Объект, который представляет заголовок страницы ★
`IsAsync`	Указывает, вызвана ли страница асинхронно ★
`IsCallback`	Указывает, загружена ли страница в ответ на обратный вызов клиентского сценария ★
`IsCrossPagePostBack`	Указывает, загружена ли страница в ответ на возврат формы, выполненный другой страницей ★
`IsPostBack`	Указывает, загружена страница в ответ на возврат формы клиентом или это её первая загрузка
`IsValid`	Указывает, успешно ли прошла валидация страницы
`Master`	Экземпляр класса `MasterPage`, представляющий эталонную страницу, связанную с текущей страницей ★
`MasterPageFile`	Имя файла эталонной страницы для текущей страницы ★

NamingContainer	Возвращает значение null[1]
Page	Возвращает текущий объект Page
PageAdapter	Возвращает адаптерный объект текущего объекта Page
Parent	Возвращает значение null
PreviousPage	В случае межстраничного возврата формы возвращает ссылку на вызывающую страницу ★
TemplateSourceDirectory	Возвращает имя виртуального каталога страницы
Validators	Коллекция имеющихся в составе страницы проверочных элементов управления
ViewStateUserKey	Пользовательский идентификатор, который предназначен для хеширования данных View State. Он используется для защиты от атак, когда злоумышленник отправляет серверу поддельное состояние представления страницы ★

В табл. 8 перечислены свойства класса Page, представляющие визуальные и невизуальные атрибуты страницы, такие как URL строки запроса, целевой клиент и таблица стилей.

Таблица 8

Свойства, специфичные для страницы

Имя свойства	Описание
ClientID	Всегда возвращает пустую строку
ClientQueryString	Возвращает URL из строки запроса ★
ClientTarget	По умолчанию содержит пустую строку; позволяет задать тип целевого браузера, для которого должна быть адаптирована результирующая разметка. При установке этого свойства отключается функция автоматического определения возможностей браузера
EnableViewState	Указывает, должна ли страница управлять данными состояния представления
EnableViewStateMac	Указывает, должна ли ASP.NET вычислять машинно-зависимый аутентификационный код и добавлять его к состоянию представления страницы
EnableTheming	Указывает, применяются ли к странице темы ★
ID	Всегда возвращает пустую строку
MaintainScrollPosition-OnPostback	Указывает, следует ли после возврата формы восстанавливать позицию просмотра страницы в браузере ★
SmartNavigation	Указывает, осуществляется ли интеллектуальная навигация[2]
StyleSheetTheme	Имя таблицы стилей, которая будет применена к странице ★
Theme	Имя темы страницы ★
Title	Возвращает и позволяет задать заголовок страницы ★
TraceEnabled	Позволяет включить или отключить трассировку страницы ★
TraceModeValue	Режим трассировки страницы ★
UniqueID	Всегда возвращает пустую строку

[1] Свойства NamingContainer и Parent имеют значение null, так как страница – это корень иерархии элементов управления.

[2] *Средствами интеллектуальной навигации* называется группа функций браузера, делающих работу пользователя со страницей более удобной. Они поддерживаются IE 5.5 и выше.

`ViewStateEncryptionMode`	Указывает, должно ли быть зашифровано состояние страницы и, если должно, то как именно
`Visible`	Указывает, должна ли ASP.NET осуществлять рендеринг страницы. Если значение равно `false`, ASP.NET не будет генерировать для страницы HTML-код, и клиент получит только тот текст, который записан в выходной поток с помощью метода `Response.Write()`

Три свойства-идентификатора (`ID`, `ClientID` и `UniqueID`) объекта `Page` всегда возвращают пустую строку. Они предназначены лишь для серверных элементов управления.

Перейдем к описанию методов класса `Page`. Все методы `Page` в соответствии с их назначением можно условно разделить на три категории. Одни из них связаны с генерированием разметки страницы, другие используются как вспомогательные при формировании страницы и управлении составляющими её элементами, третьи имеют отношение к работе с клиентскими сценариями.

В табл. 9 перечислены методы, которые так или иначе связаны с генерированием кода разметки страницы.

Таблица 9

Методы `Page`, связанные с генерированием кода разметки

Имя метода	Описание
`DataBind()`	Этот метод связывает все имеющиеся в составе страницы элементы управления с их источниками данных. Сам метод `DataBind()` не генерирует разметку, но осуществляет необходимую подготовку к последующему рендерингу
`RenderControl()`	Выводит HTML-код, в том числе информацию трассировки, если данная функция включена
`VerifyRendering-InServerForm()`	Элементы управления вызывают этот метод непосредственно перед рендерингом, проверяя, включены ли они в тело серверной формы

В табл. 10 перечислены вспомогательные методы класса `Page`, предназначенные для управления дочерними элементами управления, выполнения их проверки и разрешения URL.

Таблица 10

Вспомогательные методы класса `Page`

Имя метода	Описание
`DesignerInitialize()`	Инициализирует объект страницы, когда она отображается в дизайнере Visual Studio или другого RAD-средства
`FindControl()`	Принимает идентификатор элемента управления и ищет его в контейнере именования страницы. Поиск не производится среди дочерних элементов управления, которые сами являются контейнерами имён
`GetTypeHashCode()`	Извлекает хэш-код, сгенерированный объектом страницы во время выполнения
`GetValidators()`	Возвращает коллекцию проверочных элементов управления для определенной группы проверки ★
`HasControls()`	Определяет, содержит ли страница дочерние элементы

LoadControl()	Компилирует пользовательский элемент управления из файла .ascx, загружает его и возвращает объект Control
LoadTemptate()	Компилирует пользовательский элемент управления из файла .ascx, загружает его и возвращает заключенным внутрь экземпляра класса SimpleTemplate, реализующего интерфейс ITemplate
MapPath()	Извлекает полный физический путь, соответствующий заданному абсолютному или относительному виртуальному пути
ParseControl()	Выполняет разбор заданной строки, содержащей допустимый код разметки, и возвращает экземпляр соответствующего элемента управления. Если строка содержит несколько элементов управления, возвращается только первый. Атрибут runat может быть опущен. Метод возвращает объект типа Control, который должен быть приведен к более конкретному типу
RegisterRequires-ControlState()	Регистрирует заданный элемент управления как требующий сохранения состояния ★
RegisterRequires-PostBack()	Регистрирует заданный элемент управления как получатель уведомления об обработке события обратного вызова, даже если его идентификатор не соответствует ни одному из идентификаторов в коллекции возвращенных данных. Элемент управления должен реализовать интерфейс IPostBackDataHandler
RegisterRequires-RaiseEvent()	Регистрирует элемент управления как обработчик события возврата формы. Этот элемент управления должен реализовать интерфейс IPostBackEventHandler
Register-ViewStateHandler()	Предназначенный главным образом для внутреннего применения, этот метод устанавливает внутренний флаг, который сигнализирует о необходимости сохранить состояние представления страницы. Если не вызвать этот метод на этапе предрендеринга, состояние представления никогда не будет сохранено. Обычно данный метод вызывается только серверным элементом управления HtmlForm. Из пользовательских приложений вызывать его ни к чему
ResolveUrl()	По заданному относительному URL возвращает абсолютный, основываясь на значении свойства TemplateSourceDirectory
Validate()	Указывает проверочным элементам управления страницы на необходимость проверить введенную информацию

В табл. 11 перечислены методы класса Page, имеющие отношение к HTML и коду сценариев, вставляемых в клиентскую страницу.

Таблица 11

Методы класса Page, связанные со сценариями

Имя метода	Описание
GetCallbackEventReference()	Получает ссылку на клиентскую функцию, осуществляющую обратный вызов серверного кода ★
GetPostBackClientEvent()	Вызывает метод GetCallbackEventReference()
GetPostBackClientHyperlink()	Добавляет javascript: в начало строки, полученной от метода GetPostBackEventReference(), например: javascript:_doPostBack('CtlID', '')

`GetPostBackEventReference()`	Возвращает прототип функции клиентского сценария, осуществляющей обратный вызов серверного кода; метод принимает объект `Control` и аргумент функции и возвращает строку, подобную следующей: `_doPostBack('CtlID', '')`
`SetFocus()`	Дает указание браузеру присвоить фокус ввода заданному элементу управления ★

В табл. 12 перечислены события класса `Page`, обработчики которых может создать программист. Некоторые из событий ортогональны типичному жизненному циклу страницы (который составляют этапы инициализации, обработки возвращенных данных формы, рендеринга) и генерируются в случаях, когда в процессе принимает участие дополнительная страница.

Таблица 12

События, которые может генерировать страница

Событие	Когда генерируется
`AbortTransaction`	Отменена автоматическая транзакция, в которой участвует данная страница ASP.NET
`CommitTransaction`	Завершена автоматическая транзакция, в которой участвует данная страница ASP.NET
`DataBinding`	Для страницы вызван метод `DataBind()`
`Disposed`	Страница удаляется из памяти
`Error`	Сгенерировано необработанное исключение
`Init`	Страница инициализируется (это первый этап её жизненного цикла)
`InitComplete`	Все дочерние элементы управления и сама страница инициализированы ★
`Load`	Инициализированная страница загружается
`LoadComplete`	Загрузка страницы завершена, сгенерированы серверные события ★
`PreInit`	Непосредственно перед началом этапа инициализации страницы ★
`PreLoad`	Непосредственно перед началом этапа загрузки страницы ★
`PreRender`	Страница готова к рендерингу
`PreRenderComplete`	Непосредственно перед началом этапа рендеринга страницы ★
`SaveStateComplete`	Состояние представления страницы сохранено в памяти ★
`Unload`	Страница выгружается из памяти, но ещё не уничтожена

5.6. ЖИЗНЕННЫЙ ЦИКЛ СТРАНИЦЫ

В данном параграфе подробно рассматриваются все этапы жизненного цикла страницы ASP.NET.

Класс страницы и входящие в её состав элементы управления отвечают за выполнение запроса к этой странице и рендеринг HTML-кода для клиента. Взаимодействие между клиентом и сервером осуществляется без сохранения состояния, поскольку такова природа протокола HTTP. Однако реальным приложениям необходимо, чтобы определенные данные состояния сохранялись между последовательными вызовами одной и той же страницы. В ASP.NET имеется встроенная инфраструктура, которая сохраняет и восстанавливает состояние страницы прозрачным для приложения способом. Таким образом, не-

смотря на природу базового протокола, пользователю кажется, что он участвует в непрерывном процессе.

Иллюзия непрерывности создается подсистемой ASP.NET, отвечающей за управление состоянием представления страниц. Перед рендерингом своего контента в HTML-формате страница кодирует и сохраняет (обычно в скрытом поле), всю информацию состояния представления. Когда происходит возврат формы, состояние представления извлекается из скрытого поля, десериализуется и используется для инициализации экземпляров серверных элементов управления.

Состояние представления уникально для каждого экземпляра страницы, поскольку оно встроено в HTML-код. Благодаря этому при возврате формы элементы управления страницы инициализируются теми же значениями, какие они имели в тот момент, когда в последний раз осуществлялся рендеринг страницы для клиента. На дополнительном этапе жизненного цикла страницы происходит объединение сохраненного состояния представления с изменениями, внесенными в результате действий клиента. Таким образом, после возврата формы страница выполняется в актуальном контексте, как будто используется непрерывное соединение «точка-точка».

Данная схема выполнения страницы основана на важном предположении о том, что страница всегда выполняет возврат формы самой себе и передаёт между клиентом и сервером состояние представления.

Весь жизненный цикл страницы условно разделяют на этапы. Одними этапами можно управлять с помощью обработчиков событий, другие требуют переопределения методов. Некоторые этапы (или их составляющие) недоступны контролю разработчика, и упоминаются здесь для полноты изложения.

В жизненном цикле страницы можно выделить три главных этапа: *подготовительный, этап обработки информации*, связанной с возвратом формы, и *завершающий*. Каждый из них делится на несколько меньших шагов, на которых генерируются события. Описание жизненного цикла страницы охватывает все возможные пути её выполнения. Однако следует понимать, что описанный процесс может изменяться в зависимости от текущей ситуации: страница выполняется в первый раз, имеет место межстраничный возврат формы, обратный вызов сценария или обычный возврат формы.

1. Подготовка страницы к выполнению

Когда исполняющая среда HTTP создает экземпляр страницы для обслуживания запроса, конструктор страницы формирует дерево её элементов управления. Подготавливаются и инициализируются все дочерние элементы управления и внутренние объекты страницы, такие как объекты запроса и ответа.

В процессе обработки запроса первым делом страница выясняет причину своего запуска. Этой причиной может быть получение обычного запроса, возврат формы, межстраничный возврат формы или обратный вызов сценария. Объект страницы конфигурирует своё внутреннее состояние с учетом этой причины и подготавливает коллекцию параметров запроса. После выполнения

этого первого шага страница готова генерировать события для пользовательского кода.

1.1. Событие `PreInit`

Это событие, введенное в ASP.NET 2.0, является точкой входа жизненного цикла страницы. Когда оно генерируется, ни эталонная страница, ни тема ещё не связаны с текущей страницей. Однако позиция прокрутки страницы уже восстановлена, доступны возвращенные клиентом данные, экземпляры всех элементов управления страницы созданы и инициированы значениями свойств по умолчанию, которые определяются в исходном файле `.aspx`. Это единственный момент, когда можно программным способом задать для текущей страницы эталонную страницу и тему. Данное событие доступно только для страницы. Свойства `IsCallback`, `IsCrossPagePostback` и `IsPostback` к этому моменту уже установлены.

1.2. Событие `Init`

Теперь эталонная страница и тема уже связаны с текущей страницей, и изменить их невозможно. Метод `ProcessRequest()` класса `Page` перебирает в цикле все дочерние элементы управления, давая каждому из них шанс инициализировать свое состояние с учетом текущего контекста. У каждого дочернего элемента управления есть метод `OnInit()`, который и вызывается. Каждому дочернему элементу управления присваивается контейнер имен и идентификатор, если они не указаны в файле `.aspx`.

Событие `Init` вначале достигает дочерних элементов управления, а уж потом самой страницы. На этом этапе страница и ее элементы управления обычно начинают загрузку определенных составляющих своего состояния. Состояние представления пока еще не восстановлено.

1.3. Событие `InitComplete`

Данное событие (оно также введено в ASP.NET 2.0) сигнализирует об окончании стадии инициализации. Между событиями `Init` и `InitComplete` страница производит только одну операцию: включает режим отслеживания изменений в состоянии представления. Эта функция обеспечивает элементам управления возможность сохранить те значения, которые программно добавляются в коллекцию `ViewState`. Если для какого-то из элементов управления данная функция отключена, любые добавленные им в коллекцию `ViewState` значения между возвратами формы утрачиваются. Все элементы управления включают режим отслеживания состояния представления сразу после того, как будет сгенерировано событие `Init`, и страница - не исключение. (Ведь, по сути, она тоже является элементом управления.)

1.4. Восстановление состояния представления

Если страница обрабатывается по причине возврата формы (её свойство `IsPostBack` содержит значение `true`), восстанавливается содержимое скрытого

поля `__VIEWSTATE` клиентской страницы. В этом поле хранится состояние представления всех элементов управления, записанное туда в конце выполнения предыдущего запроса. Общее состояние представления страницы является своего рода контекстом вызова и содержит состояние всех ее элементов управления, в последний раз переданное браузеру.

На данном этапе каждый элемент управления получает возможность обновить свое текущее состояние, сделав его таким, каким оно было во время выполнения предыдущего запроса. События не генерируются, а если необходимо выполнить некую обработку, можно переопределить метод `LoadViewState()`, определенный как защищенный и виртуальный в классе `Control`.

1.5. Обработка данных, принятых в результате возврата формы

На этом этапе обрабатываются все клиентские данные, содержащиеся в запросе HTTP, то есть содержимое всех полей ввода, определенных в тэге `<form>`. Обычно данные пересылаются в такой форме:

```
TextBox1=text&DropDownList1=selectedItem&Button1=Submit
```

Это разделенная символами `&` строка пар «*имя=значение*», которые загружаются в специальную коллекцию, предназначенную для внутреннего использования. Процессор страницы ищет соответствие между именами из полученной коллекции и идентификаторами элементов управления страницы. Найдя его, он проверяет, реализует ли соответствующий серверный элемент управления интерфейс `IPostBackDataHandler`. Если да, вызываются методы этого интерфейса, чтобы дать элементу управления возможность обновить свое состояние с использованием полученных данных. В частности, процессор страницы вызывает метод `LoadPostData()`. Если этот метод возвращает `true`, то состояние элемента управления обновлено, и последний добавляется в отдельную коллекцию для дальнейшей обработки.

Если имя полученного элемента данных не принадлежит ни одному из серверных элементов управления, он игнорируется и временно помещается в отдельную коллекцию для повторной попытки идентификации, которая будет предпринята позднее.

1.6. Событие `PreLoad`

Данное событие (введено в ASP.NET 2.0) указывает, что страница завершила этап инициализации системного уровня и переходит к тому этапу, на котором пользовательский код страницы имеет возможность сконфигурировать ее для дальнейшего выполнения и рендеринга. Это событие генерируется только для страниц.

1.7. Событие `Load`

Это событие генерируется сначала для страницы, а потом рекурсивно для всех её дочерних элементов управления. К этому моменту элементы управления, составляющие дерево страницы, уже созданы, и их состояние полностью восстановлено с учетом данных, полученных от клиента. Страница готова к

выполнению инициализационного кода, связанного с её логикой и поведением. На этом этапе доступ к свойствам элемента управления и состоянию представления можно осуществлять без каких-либо опасений.

1.8. Обработка динамически созданных элементов управления

Когда все элементы управления страницы получили возможность завершить перед отображением свою инициализацию, процессор страницы предпринимает вторую попытку идентифицировать полученные от клиента значения, для которых не нашлось соответствий среди существующих элементов управления. Для этого он повторяет действия, описанные выше, в подразделе 1.5. Такое его поведение объясняется очень просто: процессор рассчитывает на то, что недостающие элементы управления могли быть созданы динамически.

Динамическое добавление элемента управления в дерево страницы может осуществляться, например, в ответ на определенное действие пользователя. Как уже упоминалось, после каждого возврата формы дерево страницы строится заново, так что информация о динамически созданных ранее элементах управления утрачивается. Однако на клиенте в момент возврата формы все динамически созданные элементы управления в ней присутствуют и заполнены данными, которые и получит сервер. Очевидно, что первоначально, то есть сразу после создания дерева страницы, полученным от клиента значениям динамических элементов управления не соответствуют никакие идентификаторы серверных элементов, но ASP.NET знает, что элементы управления могут создаваться в обработчике события `Load`. Вот потому-то и производится вторая попытка идентификации полученных значений как принадлежащих динамическим элементам управления.

2. Обработка возврата формы

Механизм возврата формы является главной движущей силой любого приложения ASP.NET. Суть операции возврата формы заключается в том, что данные формы клиентской страницы передаются серверной странице - той самой, которая эту клиентскую страницу сгенерировала, и серверная страница восстанавливает контекст вызова, используя сохраненное ранее состояние представления и текущие данные формы.

После того как страница выполнила инициализацию и обработала полученные от клиента значения, приходит время двух групп серверных событий: события первой группы сигнализируют об изменении состояния определенных серверных элементов управления, а события второй группы генерируются в ответ на действие клиента, вызвавшее возврат формы.

2.1. Обнаружение изменений в состоянии элементов управления

Система ASP.NET действует, исходя из предположения о наличии взаимно-однозначного соответствия между HTML-тэгами ввода, используемыми в браузере, и элементами управления ASP.NET, функционирующими на сервере. Примером может служить соответствие между тэгом `<input type="text">` и

элементом управления TextBox. Технически связь между этими двумя элементами устанавливается посредством их идентификаторов, которые должны быть одинаковыми.

Для всех элементов управления, вернувших из метода LoadPostData() значение true, пришло время выполнить второй метод интерфейса IPostBackDataHandler – RaisePostDataChangedEvent(). Его вызов сигнализирует элементу управления, что пора уведомить приложение ASP.NET об изменении своего состояния. Реализация данного метода оставляется на усмотрение разработчика элемента управления, однако большинство элементов делают в нем одно и то же: генерируют серверное событие и предоставляют разработчику страницы возможность включиться в игру и выполнить код, обрабатывающий данное событие. Например, если после возврата формы содержимое свойства Text элемента управления TextBox оказалось измененным, элемент управления TextBox генерирует событие TextChanged.

2.2. Обработка серверного события возврата формы

Операция возврата формы начинается с того, что на клиенте осуществляется некоторое действие, требующее реакции сервера. Например, пользователь щёлкает кнопку, предназначенную для отправки содержимого формы серверу. Такая клиентская кнопка, обычно реализованная как гиперссылка или кнопка submit-типа[1], связана с серверным элементом управления, реализующим интерфейс IPostBackEventHandler.

Процессор страницы просматривает полученные от клиента данные и определяет, какой элемент управления инициировал возврат формы. Если этот элемент реализует интерфейс IPostBackEventHandler, процессор вызывает его метод RaisePostBackEvent(). Реализация данного метода оставлена на усмотрение разработчика элемента управления и, теоретически, может у разных элементов несколько различаться. Однако на практике все элементы управления генерируют в нём серверное событие, позволяющее автору страницы программно отреагировать на возврат формы. Например, элемент управления Button генерирует событие OnClick.

2.3. Событие LoadComplete

Введенное в ASP.NET 2.0 и поддерживаемое только для страниц, событие LoadComplete сигнализирует об окончании этапа подготовки страницы. Обратите внимание, что дочерние элементы управления этого события не получают. Сгенерировав событие LoadComplete, страница вступает в фазу рендеринга.

3. Завершающий этап выполнения страницы

После обработки события возврата формы страница готова сгенерировать вывод для браузера. Этап рендеринга делится на две стадии: предрендеринг и

[1] Страница может осуществить возврат формы двумя способами: с использованием кнопки submit-типа (то есть элемента <input type="submit">) или посредством сценария.

генерирование разметки. Стадия предрендеринга также делится на две части, которым соответствуют события предобработки и постобработки.

3.1. Событие PreRender

Обрабатывая событие `PreRender`, страница и элементы управления могут выполнять изменения, которые необходимо внести до того, как начнется рендеринг страницы. Данное событие сначала достигает страницы, а затем рекурсивно всех ее элементов управления. Этот этап особенно важен для составных элементов управления.

3.2. Событие PreRenderComplete

Поскольку событие `PreRender` рекурсивно генерируется для всех дочерних элементов управления страницы, ее автор не знает, когда завершится фаза предрендеринга. Поэтому в ASP.NET 2.0 введено новое событие, `PreRenderComplete`, генерируемое только для страницы и уведомляющее об этом моменте.

3.3. Событие SaveStateComplete

Следующим шагом, предшествующим генерированию разметки клиентской страницы, является сохранение её текущего состояния. Каждое действие, выполненное после этого момента и связанное с модификацией состояния представления, может отразиться на рендеринге, но внесенные при этом изменения состояния представления уже не сохранятся и при следующем возврате формы будут утрачены. Сохранение состояния страницы - рекурсивный процесс, при выполнении которого процессор страницы проходит по всему ее дереву, вызывая метод `SaveViewState()`[1] элементов управления и самой страницы. `SaveViewState()` - это защищенный и виртуальный метод, отвечающий за сохранение содержимого словаря `ViewState` текущего элемента управления.

В ASP.NET 2.0 помимо состояния представления с элементами управления связано еще и так называемое *состояние элемента* - род приватного состояния представления, не являющегося предметом управления со стороны приложения. Иными словами, управление состоянием элемента нельзя программно отключить, как это можно сделать для состояния представления. Так вот, состояние элемента также сохраняется на данном этапе.

Событие `SaveStateComplete`, введенное в ASP.NET 2.0, генерируется после полного сохранения состояния элементов управления страницы.

---

[1] Данные состояния представления страницы и отдельных ее элементов управления накапливаются в специальной структуре памяти и затем сохраняются – по умолчанию в скрытом поле `__VIEWSTATE`. Сериализация и десериализация этих данных производится парой переопределяемых методов класса `Page`: `SavePageStateToPersistenceMedium()` и `LoadPageStateFromPersistenceMedium()`. Переопределив оба метода, вы можете, например, сохранять состояние представления в серверной базе данных или в состоянии сеанса, кардинально уменьшив размер отправляемой пользователю страницы.

3.4. Генерирование разметки

Генерирование разметки для браузера осуществляется путем вызова каждого элемента управления страницы, с тем, чтобы он сгенерировал собственную разметку и вывел ее в буфер, где накапливается код формируемой клиентской страницы. Несколько переопределяемых методов позволяют разработчику вмешиваться в процесс на разных этапах генерирования разметки: когда выводится начальный тэг, тело и конечный тэг. Пользовательские события с этапом рендеринга не связаны.

3.5. Событие `Unload`

По окончании этапа рендеринга для каждого элемента управления, а затем и для самой страницы генерируется событие `Unload`. Это событие позволяет элементам выполнить перед освобождением объекта страницы заключительные операции, такие как закрытие файлов и подключений к базам данных.

Заметим, что уведомление о выгрузке страницы или элемента управления из памяти поступает непосредственно перед выполнением этой операции, поэтому, пока происходит его обработка, объект еще не удален. Для освобождения памяти, занимаемой объектом страницы, её процессор вызывает метод `Dispose()`. Это происходит сразу после того, как завершится выполнение всех рекурсивно вызванных обработчиков события `Unload`. Переопределение метода `Dispose()` класса `Page` или обработка события `Disposed` страницы, даёт последнюю возможность выполнить для неё действия по освобождению ресурсов, пока она сама не будет удалена из памяти.

5.7. ОБЩИЙ ОБЗОР СЕРВЕРНЫХ ЭЛЕМЕНТОВ УПРАВЛЕНИЯ

Важным элементом технологии ASP.NET являются *серверные элементы управления* (*server controls*). Серверный элемент управления – это некий класс, объект которого агрегирован в страницу. При выполнении рендеринга страницы серверный элемент транслируется в один или несколько HTML-тэгов.

Библиотека серверных элементов насчитывает порядка сотни классов. Условно все элементы можно разделить на следующие группы:

1. **Элементы управления HTML** – эквиваленты обычных HTML-тэгов. Имеют атрибуты, сходные с атрибутами HTML-тэгов.
2. **Элементы управления Web** – набор элементов, равнозначный обычным HTML-элементам, но обладающих расширенным набором свойств и событий, облегчающий создание страницы.
3. **Проверочные элементы управления** – набор специальных элементов для проверки корректности значений, вводимых в другие элементы на странице. Выполняют проверку на стороне клиента и сервера.
4. **Элементы управления для представления данных** – обеспечивают различные способы представления данных, хранящихся как в локальных коллекциях, так и в базах данных.

5. **Навигационные элементы управления** – группа элементов для навигации по сайту (например, реализация меню сайта).
6. **Элементы управления Login** – элементы для работы с пользователями сайта - регистрация, вход, управление паролями и ролями.
7. **Элементы управления Web Parts** – специфические элементы управления для реализации технологии Web Parts.
8. **Элементы управления AJAX Extensions** –элементы управления для технологии AJAX (введены в ASP.NET 3.5).

Для всех серверных элементов управления справедливы следующие замечания. Настройка элементов может быть выполнена декларативно и программно. При декларативной настройке значения атрибутов элемента транслируются в соответствующие значения свойств. Например, в следующем фрагменте разметки страницы у элемента `Button` (кнопка) заданы свойства `ID` (идентификатор элемента и имя поля в классе страницы) и `Text` (надпись на кнопке):

```
<asp:Button ID="btnSend" runat="server" Text="Send" />
```

При задании серверного элемента на странице используется соответствующий тэг с обязательным атрибутом `runat="server"`.

В некоторых случаях элемент управления образует в описании страницы секцию, ограниченную открывающими и закрывающим тэгом, а свойство элемента задано как содержимое секции.

```
<asp:Label ID="lblExamlpe" runat="server">
    This is a label content
</asp:Label>
```

Абсолютно все серверные элементы управления являются наследниками класса `System.Web.UI.Control`. Рассмотрим свойства, методы и события данного класса. Свойства класса `Control` приведены в табл. 13.

Таблица 13

Свойства класса `Control`

Имя свойства	Описание
`BindingContainer`	Элемент управления, который является логическим родителем текущего элемента в контексте связывания с данными ★
`ClientID`	Идентификатор, который будет присвоен элементу управления в HTML-странице. Является измененной версией свойства `UniqueID`: символы знак доллара (`$`) в `UniqueID` заменяются символом подчеркивания (`_`) в `ClientID`
`Controls`	Коллекция ссылок на все дочерние элементы управления
`EnableTheming`	Указывает, применяются ли к элементу управления темы ★
`EnableViewState`	Указывает, должен ли элемент сохранять между запросами состояние представления (свое и своих дочерних элементов)
`ID`	Имя, которое будет использоваться для программной идентификации элемента управления в коде страницы

NamingContainer	Контейнер именования элемента управления. Контейнером именования элемента является его родительский элемент, реализующий интерфейс INamingContainer. Если такого элемента не существует, контейнером именования является страница
Page	Страница, содержащая элемент управления
Parent	Родительский элемент данного элемента управления согласно иерархии элементов страницы
Site	Контейнер, в котором содержится текущий элемент управления, когда он выводится на рабочей поверхности дизайнера. Используется в дизайнере Visual Studio
SkinID	Имя обложки, применяемой к элементу управления ★
TemplateControl	Шаблон, содержащий текущий элемент управления ★
TemplateSourceDirectory	Имя виртуального каталога хост-страницы
UniqueID	Иерархически уточненный идентификатор элемента
Visible	Указывает и позволяет определить, должна ли ASP.NET осуществлять рендеринг элемента управления

В табл. 14 приведено описанием методов класса Control.

Таблица 14

Методы класса Control

Имя метода	Описание
ApplyStyleSheetSkin()	Применяет к элементу управления свойства, определенные в таблице стилей страницы. Какая именно обложка используется, зависит от значения свойства SkinID ★
DataBind()	Генерирует событие DataBinding, после чего вызывает для всех дочерних элементов управления метод DataBind()
Dispose()	Предоставляет элементу управления возможность выполнить задачи очистки, прежде чем тот будет удален из памяти
Focus()	Присваивает элементу управления фокус ввода ★
FindControl()	Ищет заданный элемент управления в коллекции дочерних элементов
HasControls()	Указывает, имеет ли элемент управления дочерние элементы
RenderControl()	Генерирует для элемента управления вывод HTML
ResolveClientUrl()	Служит для получения URL, который может использоваться клиентом для доступа к ресурсам веб-сервера, таким как файлы изображений, дополнительные страницы и т. д.
ResolveUrl()	Разрешает относительный URL, возвращая абсолютный на основе значения свойства TemplateSourceDirectory

Класс Control определяет набор базовых событий, поддерживаемых всеми серверными элементами управления. Эти события описаны в табл. 15.

Таблица 15

События класса `Control`

Событие	Когда происходит
`DataBinding`	Для элемента управления вызван метод `DataBind()` и производится связывание этого элемента с источником данных
`Disposed`	Элемент управления удаляется из памяти (это последний этап его жизненного цикла)
`Init`	Элемент управления инициализируется
`Load`	Элемент управления загружается в память. Данное событие происходит после события `Init`
`PreRender`	Элемент управления готов к рендерингу своего содержимого
`Unload`	Элемент управления выгружается из памяти

Начиная с ASP.NET 2.0 можно декларативно присвоить всем свойствам элемента управления значения, специфичные для конкретных браузеров. Вот небольшой пример разметки:

```
<asp:Button ID="btn" runat="server" Text="A button"
            ie:Text="IE Button" mozilla:Text="Firefox Button" />
```

Свойство `Text` кнопки содержит значение `"IE Button"`, когда страница выводится в Internet Explorer, и значение `"Firefox Button"`, когда она выводится в Firefox. Если страницу запрашивает другой браузер, используется атрибут `Text` без префикса. Все свойства, которые разрешается задавать в составе тэга, можно помечать идентификатором браузера[1]. ASP.NET 2.0 поддерживает *адаптивный рендеринг*, то есть процесс генерирования разметки, производимый с учетом особенностей целевого браузера. Для поддержки адаптивного рендеринга задача генерирования разметки делегирована внешнему по отношению к элементу управления компоненту - *адаптеру*. Выбор адаптера определяется возможностями целевого браузера, сведения о которых извлекаются из базы данных браузеров ASP.NET. Если запись об определенном браузере содержит имя класса адаптера элемента управления, создается и используется экземпляр этого класса. В противном случае адаптером элемента управления становится экземпляр класса `ControlAdapter` - универсальный адаптер.

В ASP.NET 2.0 наряду с поддержкой состояния представления каждый элемент управления поддерживает *состояние элемента управления* (*Control State*). Оба состояния преследуют схожие цели – сохранение информации элемента между запросами. Основное отличие - состояние элемента управления нельзя отключить программно или декларативно (т. е. оно всегда включено). Каждый элемент управления сохраняет и загружает свое состояние, используя пару виртуальных методов `SaveControlState()` и `LoadControlState()`. Состояние элемента управления используется подобно состоянию представления, со-

[1] Для получения списка возможных идентификаторов браузера обратитесь к директории `%WINDOWS%\Microsoft.NET\Framework\[версия]\CONFIG\Browsers`

хранится и загружается на тех же этапах, что и состояние представления, и содержится в том же скрытом поле.

5.8. ЭЛЕМЕНТЫ УПРАВЛЕНИЯ HTML

ASP.NET содержит набор элементов управления, близко соответствующий обычным элементам HTML. В коде разметки страницы *серверные элементы управления HTML* выглядят как обычные HTML-теги, но с атрибутом `runat="server"`. Серверные элементы HTML определены для большинства тэгов, таких как `<form>`, `<input>`, `<select>`, `<table>`, `<img>`, `<a>`. Все они наследуются от одного класса – `HtmlControl` (пространство имен `System.Web.UI.HtmlControls`), производного от класса `Control`. Уникальные свойства класса `HtmlControl` описаны в табл. 16.

Таблица 16

Свойства класса `HtmlControl`

Свойство	Описание
`Attributes`	Коллекция всех атрибутов элемента управления и их значений
`Disabled`	Булево значение, указывающее, отключен ли элемент управления
`Style`	Коллекция, представляющая все CSS-свойства элемента управления
`TagName`	Возвращает имя HTML-тэга элемента управления

Коллекцию `Attributes` можно использовать для установки таких HTML-атрибутов, которым не соответствуют специализированные свойства элемента управления. Содержимое коллекции заносится в HTML-вывод как набор пар «`атрибут=значение`». Например, в следующем коде устанавливается значение атрибута `onload` HTML-тега `<body>`.

```
<script>
    function Init() { alert("Hello"); }
</script>

<script runat="server" language="C#">
    protected void Page_Load(object sender, EventArgs e)
    {
        theBody.Attributes["onload"] = "Init()";
    }
</script>

<html>
   <body ID ="theBody" runat="server" />
</html>
```

На рис. 1 показано дерево доступных элементов управления HTML.

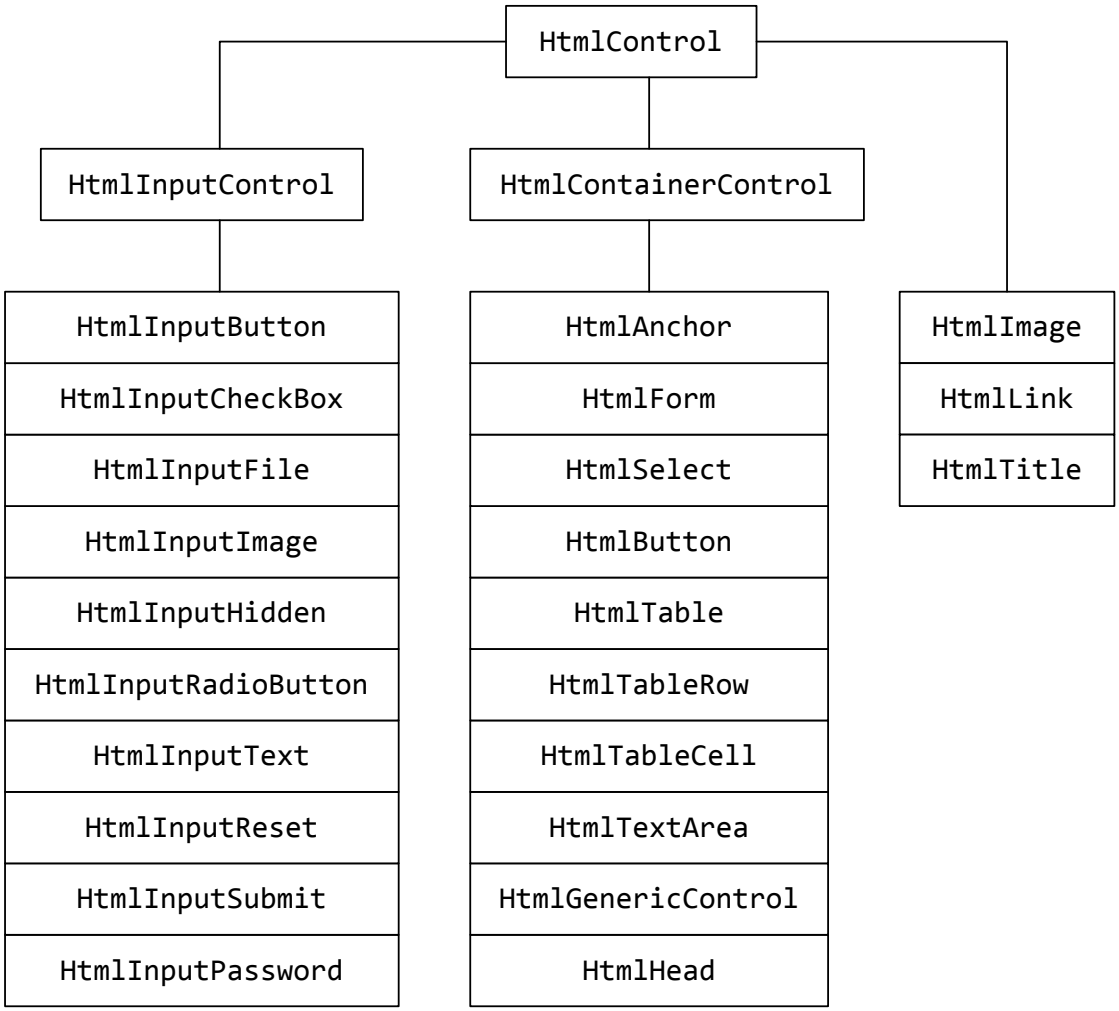

Рис. 1. Дерево элементов управления HTML

Большинство элементов управления HTML можно разделить на две категории: *контейнерные* и *предназначенные для ввода данных*. Контейнерные элементы управления HTML перечислены в табл. 17.

Таблица 17

Контейнерные элементы управления HTML

Класс	Что представляет
HtmlAnchor	Якорь HTML, тэг <a>
HtmlButton	Тэг <button>, который определен в спецификации HTML 4.0
HtmlForm	Тэг <form>. Может использоваться только в качестве контейнера интерактивных серверных элементов управления, но не пригоден для создания HTML-форм, программируемых на сервере
HtmlGenericControl	HTML-тэг, для которого в ASP.NET не определен специальный класс. Примерами являются тэги <font>, <hr>, <iframe>. Их программируют с использованием коллекции Attributes
HtmlHead	Тэг <head>. Позволяет программно управлять метатэгами, таблицами стилей и заголовком страницы ★
HtmlSelect	Тэг <select>, то есть группу вариантов выбора
HtmlTable	Таблицу HTML - тэг <table>
HtmlTableCell	Тэг <td>, то есть ячейку таблицы

HtmlTableRow	Тэг <tr>, то есть строку таблицы
HtmlTextArea	Многострочное текстовое поле - тэг <textarea>

Базовым классом контейнерных элементов управления HTML является класс HtmlContainerControl. Он представляет все элементы, которые должны иметь закрывающийся тэг: формы, блоки выбора, таблицы, а также якоря и текстовые области. По сравнению с классом HtmlControl контейнерный элемент управления имеет два дополнительных строковых свойства - InnerHtml и InnerText. Оба свойства управляют чтением и записью литерального контента, расположенного между открывающим и закрывающим тэгами элемента[1].

Обсудим некоторые возможности контейнерных элементов. Для страницы, содержащей тэг <head> с атрибутом runat="server", автоматически создается элемент управления HtmlHead, определяющий *заголовок страницы*. Заголовок страницы представлен новым свойством Header класса Page. Если тэг <head> отсутствует или не имеет атрибута runat="server", данное свойство содержит null. Элемент управления HtmlHead имеет свойство Title, посредством которого можно извлекать и задавать заголовок страницы.

Якорный элемент управления HtmlAnchor и элемент управления HtmlButton могут использоваться не только для перехода к другой странице (что является их основным назначением), но и для осуществления возврата формы. Они обладают серверным событием ServerClick. Вы можете определить имена методов, которые будут обрабатывать на сервере и клиенте событие, вызванное тем, что пользователь щелкнул элемент управления. Ниже продемонстрировано объявление якоря, в котором событию-щелчку назначены и клиентский, и серверный обработчики. Атрибутом onclick определяется клиентский обработчик, написанный на JavaScript, а атрибутом onserverclick - серверный обработчик, код которого будет выполнен после возврата формы.

```
<a runat="server" onclick="Run()" onserverclick="DoSome">Click</a>
```

Элемент управления HtmlSelect представляет список опций, из которых можно выбрать одну или несколько. Этот элемент поддерживает *связывание с источником данных* (будет рассмотрено позднее). Его поведением управляют свойства Size (количество опций) и Multiple (разрешен ли выбор нескольких опций). Сами элементы-опции хранятся в коллекции Items, состоящей из объектов ListItem. Чтобы задать текст элементов, можно либо установить свойство Text каждого из объектов ListItem, либо разместить между открывающимся и закрывающимся тэгами <select> группу тэгов <option>.

В ASP.NET простейшую таблицу HTML можно вывести с помощью элемента управления HtmlTable. Но серверные таблицы не столь мощны, как обычные таблицы HTML, создаваемые с использованием тэга <table>. Главное их ограничение: HtmlTable не поддерживает тэги <caption>, <col>, <colgroup>,

[1] Учтите, что в случае, когда внутренний контент элемента управления включает серверные элементы управления, получение литерального контента невозможно.

<tbody>, <thead> и <tfoot>. Дочерними элементами HtmlTable по определению могут быть только объекты класса HtmlTableRow[1].

Элемент управления HtmlTextArea, соответствующий тэгу <textarea>, позволяет программно создавать и конфигурировать многострочные текстовые поля. У класса HtmlTextArea имеются свойства Rows и Cols, с помощью которых задаётся количество строк и столбцов поля, а свойство Value может использоваться для определения выводимого в нем текста. При возврате формы класс HtmlTextArea генерирует событие ServerChange, которое позволяет проверить на сервере данные, содержащиеся в элементе управления.

В языке HTML элемент <input> имеет несколько разновидностей и может использоваться для вывода кнопки типа submit, флажка или текстового поля. Каждой из таких разновидностей элемента <input> соответствует свой класс ASP.NET. Все эти классы являются производными от HtmlInputControl - абстрактного класса, определяющего их общий программный интерфейс. Данный класс наследуется от HtmlControl, добавляя к нему свойства Name, Type и Value. Свойство Name возвращает имя, присвоенное элементу управления. Свойство Type, соответствующее атрибуту type HTML-элемента, доступно только для чтения. Что касается свойства Value, представляющего содержимое поля ввода, то его значение можно и считывать, и записывать.

В табл. 18 перечислены элементы управления ASP.NET, соответствующие различным разновидностям тэга <input>.

Таблица 18

Элементы управления HTML, предназначенные для ввода данных

Класс	Что представляет
HtmlInputButton	Различные виды кнопок, поддерживаемые HTML. Допустимыми значениями атрибута Type являются button, submit и reset
HtmlInputCheckBox	Флажок HTML, то есть тэг <input> типа checkbox
HtmlInputFile	Загрузчик файлов - тэг <input> типа file
HtmlInputHidden	Скрытый буфер текстовых данных - тэг <input> типа hidden
HtmlInputImage	Графическая кнопка - тэг <input> типа image
HtmlInputPassword	Защищённое текстовое поле - тэг <input> типа password ★
HtmlInputRadioButton	Переключатель - тэг <input> типа radio
HtmlInputReset	Командная кнопка типа reset ★
HtmlInputSubmit	Командная кнопка типа submit ★
HtmlInputText	Текстовое поле - тэг <input> типа password или text

Событие ServerChange, генерируемое при изменении в состоянии элемента управления между возвратами формы, поддерживается input-элементами HtmlInputCheckBox, HtmlInputRadioButton, HtmlInputHidden и HtmlInputText.

Элемент управления HtmlInputButton генерирует событие ServerClick, позволяя задать код, который будет выполнен на сервере после щелчка кнопки.

Элемент управления HtmlInputImage имеет несколько дополнительных свойств, связанных с выводом изображения. В частности, он позволяет задать

[1] В ASP.NET 2.0 для поддержки простых таблиц введён элемент управления Table.

альтернативный текст (выводимый, когда изображение недоступно), рамку и способ выравнивания изображения по отношению к остальной части страницы. Обработчик события `ServerClick` этого элемента управления получает структуру данных `ImageClickEventArgs`, которая хранит в свойствах `X` и `Y` координаты указателя мыши в момент щелчка.

Элемент управления `HtmlInputFile` является HTML-средством загрузки файлов из браузера на веб-сервер. На сервере файл упаковывается в объект типа `HttpPostedFile` и остается там до тех пор, пока не будет явно обработан, например, сохранен на диске или в базе данных. Объект `HttpPostedFile` имеет свойства и методы, с помощью которых можно получить информацию о файле, извлечь его и сохранить. Кроме этого, элемент управления `HttpInputFile` позволяет ограничить перечень типов файлов, которые разрешено загружать на сервер. ASP.NET позволяет в некоторой степени контролировать количество загружаемых данных. Максимально допустимый размер файла (по умолчанию - 4 Мбайт) можно задать в атрибуте `maxRequestLength` раздела `<httpRuntime>` конфигурационного файла веб-приложения.

5.9. ЭЛЕМЕНТЫ УПРАВЛЕНИЯ WEB

Элементы управления Web, определенные в пространстве имен `System.Web.UI.WebControls`, являются серверными компонентами, создаваемыми при наличии атрибута `runat="server"`. В теле aspx-страницы Web-элементы можно отличить по префиксу пространства имен `asp`. Некоторые из этих элементов управления подобны серверным элементам HTML, но интерфейс программирования при этом имеют разный. Web-элементы обладают более согласованным и абстрактным API и более богатой функциональностью.

Все элементы управления Web наследуются от базового класса `WebControl`. Он является производным от класса `Control` и определяет ряд собственных свойств и методов. Большинство членов класса `WebControl` связаны с внешним видом и поведением элементов управления (это шрифты, стиль, цвета, CSS).

Свойства класса `WebControl` перечислены в табл. 19.

Таблица 19

Свойства класса `WebControl`

Свойство	Описание
`AccessKey`	Позволяет задать клавишу, которая совместно с клавишей `Alt` будет использоваться для быстрого перехода к элементу управления. Поддерживается в IE версии 4.0 и выше
`Attributes`	Коллекция атрибутов, не имеющих соответствия среди свойств элемента управления. Атрибуты, задаваемые через эту коллекцию, выводятся в составе страницы как атрибуты HTML
`BackColor`	Цвет фона элемента управления
`BorderColor`	Цвет рамки элемента управления
`BorderStyle`	Стиль рамки элемента управления
`BorderWidth`	Ширина рамки элемента управления
`CssClass`	Класс CSS, связанный с элементом управления

Enabled	Указывает, активен ли элемент управления
Font	Свойства шрифта элемента управления
ForeColor	Цвет фона элемента управления; используется при выводе текста
Height	Высота элемента управления. Задается как значение типа Unit
Style	Возвращает коллекцию CssStyleCollection, составленную из атрибутов, которые присвоены выводимому тэгу элемента
Tablndex	Индекс перехода по клавише Tab для элемента управления
ToolTip	Текст всплывающей подсказки, которая выводится при наведении на элемент управления указателя мыши
Width	Ширина элемента управления. Задается как значение типа Unit

Класс WebControl определяет несколько дополнительных методов, отсутствующих у базового класса Control. Все они перечислены в табл. 20.

Таблица 20

Методы класса WebControl

Метод	Описание
ApplyStyle()	Копирует в элемент управления непустые элементы заданного стилевого объекта. Существующие стилевые свойства переопределяются
CopyBaseAttributes()	Импортирует из заданного элемента управления Web свойства AccessKey, Enabled, ToolTip, TabIndex и Attributes. Иными словами, копирует все свойства, не инкапсулированные объектом Style
MergeStyle()	Подобно методу ApplyStyle() копирует в элемент управления непустые элементы заданного стиля, но существующие стилевые свойства не переопределяются
RenderBeginTag()	Осуществляет рендеринг открывающегося HTML-тэга элемента управления в заданный объект записи текста. Вызывается перед методом RenderControl()
RenderEndTag()	Осуществляет рендеринг закрывающегося HTML-тэга элемента управления. Вызывается сразу после метода RenderControl()

Перечислим наиболее популярные и важные элементы управления Web в табл. 21, а затем рассмотрим некоторые из них более подробно.

Таблица 21

Ключевые элементы управления Web

Элемент управления	Что представляет
Button	Кнопку, реализованную в виде тэга <input>
Checkbox	Флажок, реализованный в виде тэга <input>
FileUpload	Элемент интерфейса, дающий возможность пользователю выбрать файл для загрузки на сервер ★
HiddenField	Скрытое поле ★
HyperLink	Якорный тэг <a>; позволяет указать либо адрес для перехода, либо сценарий для выполнения
Image	Изображение, реализованное в виде тэга <img>
ImageButton	Изображение, отвечающее на щелчки мыши подобно настоящей кнопке

ImageMap	Изображение с необязательной областью в нём, которую можно щелкать мышью ★
Label	Обыкновенный статический текст, не реагирующий на щелчки. Реализован в виде тэга
LinkButton	Якорный тэг, обеспечивающий возврат формы с использованием соответствующего механизма ASP.NET. Это гиперссылка особого рода, для которой программист не может задать целевой URL
MultiView	Элемент управления, действующий как контейнер группы дочерних элементов типа View ★
Panel	HTML-контейнер, реализованный с использованием блочного элемента . В ASP.NET 2.0 этот контейнер поддерживает скроллинг
RadioButton	Одну кнопку переключателя, реализованную в виде тэга <input>
Table	Внешний табличный контейнер; эквивалентен HTML-элементу <table>
TableCell	Ячейку таблицы; эквивалентен HTML-элементу <td>
TableRow	Строку таблицы; эквивалентен HTML-элементу <tr>
TextBox	Текстовое поле, реализованное в виде тэга <input> или <textarea>, что зависит от запрошенного типа текста. Может работать в одно- или многострочном режиме либо в режиме ввода пароля
View	Контейнер группы элементов управления. Этот элемент управления всегда должен содержаться в элементе управления MultiView ★

Кнопочные элементы управления

В ASP.NET элементы управления Web, генерирующие кнопки, реализуют интерфейс IButtonControl. Его реализуют элементы Button, ImageButton и LinkButton, а в общем случае - любой специализированный элемент управления, который должен действовать как кнопка. В табл. 22 перечислены все члены интерфейса IButtonControl.

Таблица 22

Интерфейс IButtonControl

Элемент	Описание
CausesValidation	Значение булева типа, указывающее, должна ли по щелчку элемента управления выполняться валидация формы
CommandArgument	Возвращает и позволяет задать значение необязательного параметра, передаваемого обработчику события Command кнопки вместе со связанным с этой кнопкой значением CommandName
CommandName	Возвращает и позволяет задать имя связанной с кнопкой команды, передаваемое обработчику события Command
PostBackUrl	Определяет URL страницы, которая будет обрабатывать возврат формы, вызванный щелчком кнопки. Данная функция, специфическая для ASP.NET, называется *межстраничным возвратом формы*
Text	Надпись на кнопке
ValidationGroup	Имя проверочной группы, в состав которой входит кнопка
Visible	Указывает, видим ли элемент управления

В дополнение к свойствам интерфейса IButtonControl, класс Button имеет свойства OnClientClick и UseSubmitBehavior. Свойство OnClientClick позволяет установить имя функции JavaScript, которая будет выполняться на клиенте в

ответ на событие onclick (свойство OnClientClick имеется также у элементов LinkButton и ImageButton).

Гиперссылки

Элемент управления HyperLink создает ссылку на другую страницу и обычно выводится в виде текста, задаваемого в свойстве Text. В качестве альтернативы гиперссылка может быть представлена изображением, и тогда URL этого изображения задается в свойстве ImageUrl. Когда установлены оба свойства, преимущество имеет ImageUrl, а содержимое свойства Text выводится в виде всплывающей подсказки. Свойство NavigateUrl определяет URL, на который указывает гиперссылка. А в свойстве Target задается имя окна или фрейма, где будет выводиться контент, расположенный по целевому URL.

Статические изображения и графические кнопки

Элемент управления Image выводит на веб-странице обычное статическое изображение, путь к которому задается в свойстве ImageUrl. Адреса изображений могут быть абсолютными или относительными. При желании в свойстве AlternateText можно задать альтернативный текст, который будет выводиться в случае, когда изображение недоступно или когда браузер не показывает изображения. Способ выравнивания изображения относительно других элементов страницы указывается в свойстве ImageAlign.

Если нужно перехватывать щелчки изображения, воспользуйтесь вместо элемента управления Image элементом ImageButton. Класс ImageButton расширяет класс Image событиями Click и Command, генерируемыми в ответ на щелчок. Обработчик событиям Click, получает структуру данных ImageClickEventArgs. Эта структура содержит информацию о координатах точки элемента управления, которую щелкнул пользователь.

Нововведением ASP.NET 2.0 стал элемент управления ImageMap. В своей простейшей форме этот элемент выводит на странице изображение. Однако когда для него определена так называемая *горячая область*, элемент управления инициирует возврат формы или переход по заданному URL. Горячей областью называется часть изображения, щелчок которой вызывает определенное действие. Она реализуется в виде класса, наследующего класс HotSpot. Существует три предопределенных типа горячих областей: многоугольники, круги и прямоугольники.

Панели с прокруткой

Элемент управления Panel служит для группировки других элементов управления с использованием тэга . В ASP.NET панели могут иметь вертикальные и горизонтальные полосы прокрутки, реализованные с использованием CSS-стиля overflow. Следующий пример показывает, как создается прокручиваемая панель:

```
<asp:Panel ID="MainPanel" runat="server" Height="60px"
           Width="420px" ScrollBars="Auto" BorderStyle="Solid">
```

```
    <h2>Choose</h2>
    <asp:CheckBox ID="cbx1" runat="server" /><br />
    <asp:CheckBox ID="cbx2" runat="server" /><br />
    <asp:CheckBox ID="cbx3" runat="server" /><br />
    <asp:CheckBox ID="cbx4" runat="server" /><br />
</asp:Panel>
```

Загрузка файлов на сервер

Элемент управления `FileUpload` обладает той же функциональностью, что и рассмотренный ранее элемент управления `HtmlInputFile`. Однако его программный интерфейс несколько иной, пожалуй, более интуитивный. Свойство `HasFile` и метод `SaveAs()` скрывают ссылку на объект, представляющий загруженный на сервер файл, а свойство `FileName` содержит имя этого файла.

Элемент управления `Xml`

Элемент управления `Xml` используется для вывода на странице ASP.NET XML-документа. Содержимое XML-файла может выводиться в исходном виде или с применением к нему XSL-трансформации (XSLT). Данный элемент управления является декларативным аналогом класса `XslTransform` и может использовать этот класс для своих целей.

С помощью элемента управления `Xml` удобно создавать блоки используемых клиентом XML-данных, задавая документы и применяемые к ним трансформации. XML-документ можно задать разными способами: используя объектную модель XML, в виде строки или указав имя файла. Трансформация XSLT определяется как заранее сконфигурированный экземпляр класса `XslTransform` или путем указания имени файла:

```
<asp:xml ID="xmlElem" runat="server" DocumentSource="document.xml"
                      TransformSource="transform.xsl" />
```

Если вы намерены применять к XML-данным ту или иную трансформацию, её можно задать между открывающим и закрывающим тэгами элемента управления. Также элемент управления `Xml` позволяет указывать требуемую трансформацию программными средствами.

Элемент управления `PlaceHolder`

Элемент управления `PlaceHolder` наследуется непосредственно от класса `Control` и используется только как контейнер для других элементов управления страницы. `PlaceHolder` не генерирует собственного вывода, и его функции ограничены отображением дочерних элементов управления, динамически добавляемых в его коллекцию `Controls`. Такой элемент управления не сообщает странице новой функциональности, а просто помогает сгруппировать связанные между собой элементы управления и облегчает их идентификацию.

Элементы управления View и MultiView

В ASP.NET 2.0 введены два новых элемента управления, предназначенных для создания группы сменяющих одна другую панелей дочерних элементов. Элемент управления `MultiView` определяет группу представлений, создаваемых экземплярами класса `View`. В каждый конкретный момент времени только одно из них активно и выводится для клиента. Элемент управления `View` предназначен для использования в составе элемента управления `MultiView`, а сам по себе он использоваться не может. Вот как определяется элемент управления `MultiView`:

```
<asp:MultiView ID="Tables" runat="server">
    <asp:View ID="Employees" runat="server">
        (здесь какие-то элементы управления)
    </asp:View>
    <asp:View ID="Products" runat="server">
        (здесь другие элементы управления)
    </asp:View>
</asp:MultiView>
```

Выбрать активное представление можно, используя событие возврата формы. Чтобы указать, какое представление будет следующим, можно либо установить свойство `ActiveViewIndex`, либо передать объект-представление методу `SetActiveView()`.

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 клиен-

том 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 провероч-

ных 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`, `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`.

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

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 1.0 проверка элемента управления происходила по принципу «все или ничего». Например, если на форме имеется набор полей и проверочных элементов, а также две кнопки, какую бы из них ни щелкнул пользователь, всегда будут проверяться все элементы управления. Иными словами, нельзя проверить одни элементы управления с помощью одной кнопки, а другие - с помощью другой. Воспользовавшись свойством `CausesValidation` кнопки, можно вовсе отключить проверку, но это не решает проблемы.

В ASP.NET 2.0 введено новое свойство - `ValidationGroup`. Оно имеется у проверочных элементов, элементов управления `input`-типа и кнопок. Пользо-

ваться 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.

5.11. ЭТАЛОННЫЕ СТРАНИЦЫ И ТЕМЫ

Для простого пользователя сети Интернет основное отличие одного сайта от другого заключается в разнообразном дизайне страниц. Прежде чем встретится с функциональными возможностями веб-приложения, посетитель сайта оценивает, нравится ли ему внешний вид страниц. Поэтому дизайн страниц является едва ли менее важным, чем общая функциональность.

Задача разработчика писать код, а задача художника-дизайнера заниматься внешним оформлением и верстkой HTML-кода страниц. При этом в большинстве случаев необходимо обеспечить возможность параллельной работы над кодом приложения и HTML-шаблоном. С этим весьма успешно справлялась технология Code Behind. Но при этом при работе над каждой страницей разработчику приходится - так или иначе - сталкиваться с разметкой страницы.

Стоит заметить, что большинство современных сайтов имеют сходный внешний вид всех страниц, и каждая страница имеет общие элементы дизайна. В ASP.NET 1.0 общие элементы дизайна заключали в пользовательские элементы управления и включались в каждую страницу. Либо, наоборот, страницы преобразовывали в элементы управления, используя одну страницу в качестве основы, в которую загружали элементы управления в зависимости от URL.

ASP.NET 2.0 предлагает новый подход разработки единообразных страниц, осованный на понятии эталонной страницы. *Эталонная страница* (*Master page*) представляет собой обычную страницу ASP.NET, обладающую несколь-

кими дополнительными свойствами и содержащую один или несколько специальных элементов управления ContentPlaceHolder. Эти элементы определяют области, в которые обычные страницы помещают свое содержимое.

Рассмотрим пример файла эталонной страницы MainMaster.master.

```
<%@ Master Language="C#" CodeFile="MainMaster.master.cs"
           Inherits="MainMaster_master" %>
<html>
<head>
    <title> My Homepage</title>
</head>
<body>
  <table width="100%">
    <tr>
      <span ID="PageTitle" runat="server" />
    </tr>
    <tr>
      <table width="100%">
        <tr>
          <td>
           <asp:ContentPlaceHolder ID="PageMenu" runat="server"/>
          </td>
          <td>
            <form ID="Form1" runat="server">
               <asp:ContentPlaceHolder ID="Content" runat="server"/>
            </form>
          </td>
        </tr>
      </table>
    </tr>
  </table>
</body>
</html>
```

Эталонная страница MainMaster.master содержит два элемента управления ContentPlaceHolder (PageMenu и Content), содержимое которых будут наполнять отдельные страницы ASP.NET. Элемент управления ContentPlaceHolder позволяет определить содержимое по умолчанию, которое будет использоваться в случае, если страница не переопределит содержимое элемента управления.

```
<asp:ContentPlaceHolder ID="PageMenu" runat="server">
    <ul>
        <li><a href="Default.aspx">Main Page</a></li>
        <li><a href="Contents.aspx">Content</a></li>
    </ul>
</asp:ContentPlaceHolder>
```

Также как и для обычной страницы, в файле программной логики эталонной страницы можно работать с имеющимися элементами управления, изменять страницу и её поведение, основываясь на параметрах запроса[1].

Использование эталонных страниц налагает особые требования на страницы, которые связаны с ними. Поскольку шаблон содержит элементы управления `ContentPlaceHolder`, то связанная страница должна содержать элементы управления `Content`, содержащие код разметки и другие элементы управления, которые будут отображаться на результирующей странице. На странице не должно быть каких-либо серверных элементов управления или кода разметки вне элементов управления `Content`.

```
<%@ Page Language="C#" CodeFile="Default.aspx.cs"
       Inherits="Default_aspx" MasterPageFile="MainMaster.master" %>

<asp:Content ID="MyMenu" runat="server"
             ContentPlaceHolderID="PageMenu">
    <ul>
        <li><a href="Page1.aspx">Page 1</a></li>
        <li><a href="Page2.aspx">Page 2</a></li>
    </ul>
</asp:Content>
<asp:Content ID="MyContent" runat="server"
             ContentPlaceHolderID="Content">
    <asp:TextBox ID="txtName" runat="server" />
    <asp:Button ID="btnShow" runat="server" />
</asp:Content>
```

Код логики страницы создается обычным образом. Единственное отличие в том, что страница не имеет собственных объектов вроде `HeadControl`, поэтому нужно использовать ссылку на страницу шаблона через свойство `Master`.

```
protected void Page_Load(object sender, EventArgs e)
{
    Master.Page.Header.Title = "Homepage";
}
```

Для того чтобы привязать страницу к шаблону используется атрибут `MasterPageFile` директивы `@Page`. Если необходимо привязать один и тот же шаблон ко всем страницам в директории, то можно задать базовый шаблон в файле `web.config` (представлен фрагмент файла `web.config`).

```
<system.web>
    <pages master="MainMaster.master" />
</system.web>
```

[1] Эталонная страница может содержать обработчики событий, подобные `Page_Load`. При связывании со страницей контента, эталонная страница выступает в виде своеобразного вложенного пользовательского элемента управления, и вызов обработчиков ее событий выполняется как для пользовательских элементов.

Кроме того, ASP.NET позволяет устанавливать эталонную страницу программным образом. Если необходимо сменить шаблон оформления страницы, необходимо делать это в обработчике события `PreInit`.

```
protected void Page_PreInit(object sender, EventArgs e)
{
   Page.MasterPageFile = "AnotherMaster.master";
}
```

Поскольку эталонная страница - это подмножество страницы, то допустимо создавать вложенные шаблоны, указывая для эталонной страницы в директиве `@Master` путь к другому шаблону с помощью атрибута `MasterPageFile`. Для этого необходимо в основном шаблоне определить элементы управления `ContentPlaceHolder`, а в «дочерних» шаблонах наряду с `ContentPlaceHolder` определить элементы управления `Content` для замещения содержимого элементов `ContentPlaceHolder` базового шаблона.

Новой возможностью ASP.NET 2.0 является использование тем. Темы позволяют задать единообразное оформление всех элементов управления, выступая неким аналогом «супер-CSS». При работе с темами используется следующая терминология:

- *Обложка* (*skin*) - именованный набор свойств и шаблонов, который можно применить к любому количеству элементов управления страницы. Обложка всегда связана с определенным типом элементов управления.
- *Таблица стилей* (*style sheet*) - CSS или серверная таблица стилей, которая может использоваться страницами сайта.
- *Тема таблицы стилей* (*style sheet theme*) - тема, используемая для абстрагирования свойств элементов управления от самих этих элементов. Элемент управления может переопределить такую тему.
- *Настроечная тема* (*customization theme*) - тема, используемая для абстрагирования свойств элементов управления от самих этих элементов. Она переопределяет любые установки, заданные в объявлении элемента управления и в определении темы таблицы стилей.

Поясним разницу между настроечными темами и темами таблиц стилей. Первые применяются для окончательной настройки сайта, и заданные в них установки переопределяют установки свойств элементов управления, содержащиеся в исходных файлах `.aspx`. Изменив тему страницы, вы полностью меняете ее внешний вид, не корректируя ни строчки в ее исходном файле. Поэтому в случае применения настроечных тем страница `.aspx` может содержать для каждого элемента управления минимум разметки. Темы таблиц стилей применяются к элементам управления сразу после их инициализации, но до того как будут применены атрибуты, заданные в файле `.aspx`. Используя тему таблицы стилей, разработчик определяет значения свойств элементов управления по умолчанию, которые затем могут быть переопределены установками, заданными в ис-

ходном файле `.aspx`. Настроечные темы всегда имеют приоритет перед темами таблиц стилей[1].

Физически тема представляет собой совокупность файлов и папок, хранящихся в одной корневой папке, имя которой является именем этой темы. Темы могут быть глобальными и локальными. *Глобальные темы* видимы всем приложениям, установленным на серверном компьютере, а локальные - только тому приложению, в котором они определены. Каталоги глобальных тем расположены по адресу:

```
%WINDOWS%\Microsoft.NET\Framework\[версия]\ASP.NETClientFiles\Themes
```

Локальные темы находятся во вложенных каталогах папки `App_Themes` приложения. В общем случае тема может содержать следующие ресурсы.

1. Файлы CSS. Файл CSS содержит заданные в специальном формате определения визуальных стилей, применяемых к элементам HTML-документа. Он располагается в корневой папке темы.

2. Файлы обложек. Файл обложки содержит специфическую для заданной темы разметку определенного набора элементов управления. Он состоит из последовательности определений элементов управления, содержащих значения большинства их визуальных свойств, и шаблонов. Каждая обложка связана с конкретным типом элементов управления и имеет уникальное имя. Для одного типа элементов управления можно определить несколько обложек. Когда к элементу управления применяется обложка, его разметка, содержащаяся в файле `.aspx` страницы, модифицируется с учетом установок обложки. Файлы обложек содержатся в корневой папке темы.

3. Файлы изображений. Элементы управления с богатым интерфейсом часто содержат изображения. Изображения, входящие в состав обложки, обычно хранятся в подпапке `Images` папки ее темы (имя подпапки можно изменить).

4. Шаблоны. Действие обложки элемента управления может распространяться не только на его визуальные свойства, но и на структуру, при условии, что данный элемент управления поддерживает шаблоны. Это позволило бы вам, меняя определение шаблона в теме, изменять внутреннюю структуру элемента управления, не затрагивая, однако, его программный интерфейс и поведение. Шаблон определяется как компонент обложки элемента управления и хранится в файле обложки.

Файл обложки напоминает обычную страницу ASP.NET и содержит объявления элементов управления и директивы импорта. Объявление элемента управления определяет, как он будет выглядеть по умолчанию. Для примера рассмотрим фрагмент файла обложки:

```
<%-- Это пример обложки элемента Button --%>
```

[1] В тексте действует следующее соглашение: если не указано иное, под словом «тема» понимается настроечная тема, а когда речь идет о теме таблицы стилей, термин приводится полностью.

```
<asp:Button runat="server"
            BorderColor="darkgray" Font-Bold="true"
            BorderWidth="1px" BorderStyle="outset"
            ForeColor="DarkSlateGray" BackColor="gainsboro" />
```

Если применить данную обложку к странице, каждый её элемент управления `Button` будет выводиться в соответствии с установками, заданными в приведённой разметке. Тема может содержать несколько обложек для определенного типа элементов управления, и все эти обложки, кроме используемой по умолчанию, должны быть именованными. Уникальное имя обложки задается в ее атрибуте `SkinID`. Установка обложки конкретного элемента управления в файле страницы `.aspx` выполняется с помощью свойства `SkinID` этого элемента. Значение данного свойства должно соответствовать имени существующей обложки текущей темы. Если тема страницы не содержит обложки с заданным именем, к элементу управления будет применена обложка, используемая по умолчанию.

Темы можно применять на разных уровнях: приложения, папки и отдельной страницы. Выбор темы на уровне приложения затрагивает все его страницы и элементы управления. Соответствующая установка задается в файле `web.config` приложения:

```
<system.web>
    <pages theme="RedTheme" />
</system.web>
```

Атрибут `theme` определяет настроечную тему, атрибут `styleSheetTheme` - тему таблицы стилей. Аналогичным образом задается тема, применяемая ко всем страницам в определенной папке: приведенную выше установку включают в содержащийся в этой папке файл `web.config`. Наконец, тему можно задать для конкретной страницы, чтобы только к ней применялись входящие в состав этой темы стили и обложки. Для того чтобы связать тему с определенной страницей, нужно установить атрибут `Theme` или `StyleSheetTheme` директивы `@Page`. Имейте в виду, что имя выбранной темы должно соответствовать имени существующего подкаталога папки `App_Themes` приложения либо имени глобальной темы. Если существуют две темы с заданным именем, локальная и глобальная, преимущество отдается локальной теме.

Инфраструктура поддержки тем ASP.NET предоставляет разработчику булево свойство `EnableTheming`, с помощью которого можно отключить поддержку обложек элементом управления и его дочерними элементами или поддержку тем страницей. По умолчанию это свойство содержит значение `true`, то есть тема, если таковая задана, используется. Свойство `EnableTheming` определено в классе `Control` и наследуется всеми серверными элементами управления и всеми страницами. Если хотите отключить поддержку тем для всех элементов управления страницы, установите в ее директиве `@Page EnableTheming=false`. Программным способом свойство `EnableTheming` статических элементов управления (тех, которые объявлены в исходном файле страницы `.aspx`) может быть

установлено только в обработчике события Page_PreInit. Для динамически же создаваемых элементов управления данное свойство должно быть установлено до того, как элемент будет добавлен в дерево элементов управления страницы.

Темы можно применять к страницам динамически, но выполнение данной задачи требует аккуратности. Исполняющая среда ASP.NET загружает информацию о теме сразу после события PreInit. Когда оно генерируется, имя темы, заданное в директиве @Page, уже известно, и именно оно будет использоваться, если вы не переопределите его в обработчике Page_PreInit.

5.12. НЕКОТОРЫЕ ПРИЁМЫ РАБОТЫ СО СТРАНИЦАМИ

В данном параграфе рассматриваются некоторые типовые ситуации и сценарии, возникающие при разработке веб-приложений.

Класс HtmlForm

Класс HtmlForm является производным от HtmlContainerControl и обеспечивает разработчику программный доступ к HTML-элементу <form>, осуществляемый посредством свойств, перечисленных в табл. 24.

Таблица 24

Специфичные свойства класса HtmlForm

Свойство	Описание
DefaultButton	Строка с идентификатором кнопочного элемента управления, который будет используемым по умолчанию элементом ★
DefaultFocus	Строка с идентификатором кнопочного элемента управления, который получит фокус ввода при отображении формы ★
Disabled	Булево значение, указывающее, отключена ли форма. Соответствует HTML-атрибуту disabled
EncType	Возвращает и позволяет задать тип кодировки. Соответствует HTML-атрибуту enctype
Method	Значение, указывающее, как браузер возвращает данные формы серверу. По умолчанию это свойство имеет значение POST; при желании ему можно присвоить значение GET
Name	Возвращает значение свойства UniqueID
SubmitDisabledControls	Указывает, следует ли отключенным на клиенте элементам управления предоставлять свои значения для включения в состав возвращаемых данных. По умолчанию - false ★
TagName	Возвращает значение "form"
Target	Имя фрейма или окна для рендеринга сгенерированного для страницы HTML-кода

Форма всегда должна иметь уникальное имя. Если программист никак ее не назвал, ASP.NET сама генерирует имя по встроенному алгоритму. Идентификатор формы можно задать в свойстве ID либо в свойстве Name. Когда установлены оба эти свойства, предпочтение отдается первому[1].

[1] Использование атрибута Name противоречит правилам XHTML.

В составе одной страницы не может быть нескольких видимых серверных форм. То есть, на странице можно разместить несколько элементов `<form>` с атрибутом `runat="server"`, но только у одного из них свойство `Visible` может быть равно `true`. Допускается наличие в составе страницы серверной формы и обычной HTML-формы (т. е. без атрибута `runat="server"`).

Перенаправление пользователя

При разработке приложений ASP.NET типичной является задача перенаправления пользователя на другую страницу. Например, в обработчике события `Click` кнопки нужно проверить введенные пользователем данные регистрации и направить его на страницу «Успешная регистрация». Как правило, для этих целей используют метод `Redirect()` объекта `Response`:

```
protected void btnRegister_Click(object sender, EventArgs e)
{
    if (IsUserDataCorrect())
    {
        Response.Redirect("~/success.aspx");
    }
}
```

Метод `Response.Redirect()` может перенаправить пользователя на любой URL, не обязательно на страницу текущего веб-приложения. Кстати, обратите внимание, что в коде обработчиков событий и в атрибутах серверных элементов управления ASP.NET для указания корня веб-приложения можно использовать символ `"~"`.

Альтернативой методу `Response.Redirect()` может служить метод `Server.Transfer()`. При использовании этого метода переход осуществляется на сервере: URL новой страницы не отражается в строке адреса браузера, а если за вызовом метода `Transfer()` следует еще какой-либо код, он никогда не выполняется. Ограничение метода `Transfer()` – переход возможен только на страницы внутри одного веб-приложения.

Межстраничный постинг

В ASP.NET 2.0 появился механизм, позволяющий изменить стандартный цикл обработки страницы - возврат данных самой себе. В ASP.NET 2.0 те элементы управления страницы, которые реализуют интерфейс `IButtonControl`, можно сконфигурировать таким образом, чтобы они осуществляли возврат формы другой странице. Это называется *межстраничным постингом*.

Для межстраничного постинга у кнопочных элементов управления изменяется свойство `PostBackUrl`. В нем задается URL целевой страницы.

```
<asp:button ID="btnPostBacks" runat="server" Text="Click"
            PostBackUrl="target.aspx" />
```

Для ссылки на страницу, выполнившую отправку, и доступа ко всем ее элементам управления используется свойство `PreviousPage`. Вот код целевой страницы, извлекающий содержимое текстового поля формы:

```
protected void Page_Load(object sender, EventArgs e)
{
    // находим элемент на предыдущей странице
    var tbxControl = (TextBox)PreviousPage.FindControl("tbxName");
}
```

Используя свойство `PreviousPage` класса `Page`, можно обращаться к любым элементам управления страницы, выполнившей постинг. Доступ к элементам управления осуществляется слабо типизированным и косвенным способом, через метод `FindControl()`. Дело в том, что целевой странице не известен тип исходной. `PreviousPage` объявлено как свойство типа `Page`, а потому оно не может предоставить доступ к странице производного класса. Если страница была вызвана сама по себе, а не является объектом межстраничного перехода, свойство `PreviousPage` возвращает `null`.

Свойство `IsCrossPagePostBack` класса `Page` возвращает `true`, если текущая страница вызвала другую страницу ASP.NET. Разумеется, для целевой страницы оно всегда возвращает `false`.

Обработка ошибок

Если в процессе работы страницы было сгенерировано необработанное исключение, ASP.NET показывает «желтый экран смерти» - стандартную страницу, содержащую информацию об ошибке. Естественно, в реальных приложениях желательно изменить данное поведение (хотя бы с точки зрения дружественного дизайна).

ASP.NET предлагает разработчикам две глобальные точки перехвата и 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.

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. Он будет перехватывать все необработанные исключения и перенаправлять их определенной странице ошибок. Такой обработчик реализуется в точности так же, как обработчик ошибок страницы. Добавьте в приложение файл `global.asax` и заполните кодом предопределенную заглушку `Application_Error` (подробнее о файле `global.asax` будет рассказано далее).

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.

Какое бы значение не было выбрано для атрибута `mode`, стандартные страницы ASP.NET с сообщениями об ошибках не порадуют информативностью. 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`[1], 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.

Перенаправление пользователей к единой для всех ошибок странице - не единственная возможность, которую предоставляет ASP.NET; эта система 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.

5.13. СВЯЗЫВАНИЕ С ДАННЫМИ

Под *связыванием данных* (*data binding*) будем понимать помещение данных из некоего источника в элемент управления на странице. При разработке страницы применение связывания подразумевает два этапа:

- указание на странице или в свойстве элемента управления источника данных;
- собственно связывание, то есть перенос данных в элемент управления.

Элементы управления, поддерживающие связывание, имеют в своем составе метод `DataBind()` для выполнения связывания. Вызов метода `DataBind()` у родительского элемента управления автоматически ведет к вызову этого метода у дочерних элементов. В частности, вызов `DataBind()` страницы обеспечивает связывание для всех её элементов.

Будем выделять два вида связывания данных: одиночное и итеративное. *Итеративное связывание* подразумевает связывание элемента управления с источником, содержащим некоторую коллекцию данных. Элементы управления,

[1] Как правило, пользовательская страница с сообщением об ошибке состоит из чистого HTML, так что больше никакая ошибка произойти не может.

пригодные для итеративного связывания, как правило, представляют собой разновидности списков или таблиц. Все они имеют общие свойства, перечисленные в табл. 25.

Таблица 25

Общие свойства, относящиеся к связыванию данных

Свойство	Описание
`DataSource`	Задаёт источник данных. В роли источника может выступать любой перечислимый объект. При вызове `DataBind()` выполняется проход по коллекции `DataSource`, и данные перемещаются в *локальное хранилище* элемента управления, чтобы затем использоваться для построения разметки элемента
`DataSourceID`	Позволяет указать идентификатор объекта-источника данных (о таких объектах будет подробнее рассказано ниже) ★
`DataMember`	При помощи свойства конкретизируется коллекция данных, в том случае, если `DataSource` обладает несколькими пригодными для связывания данных наборами. (типичный пример: `DataSource` содержит рассоединённый набор данных `DataSet`, а `DataMember` указывает на таблицу для связывания)
`DataTextField`	Имя столбца источника данных, в котором содержатся значения, подлежащие выводу в качестве элементов списка. Обычно используется списочными элементами управления
`DataValueField`	Определяет, какое поле источника данных надо использовать для заполнения набора данных списочного элемента управления
`AppendDataBoundItems`	Булево свойство; указывает, нужно ли добавлять загружаемые в элемент управления новые данные в конце существующего набора его данных ★
`DataKeyField`	Ключевое поле источника данных. Оно используется табличными элементами управления ASP.NET 1.0 и позволяет им идентифицировать запись

При выполнении связывания данных ASP.NET позволяет указывать в разметке страницы специальные выражения связывания с данными. *Выражения связывания с данными* - это исполняемый код, заключенный внутрь специальной конструкции `<%  %>` и начинающийся с символа-префикса `#`. Обычно такие выражения используются для установки значения атрибута в тэге серверного элемента управления. Например, в следующем фрагменте кода разметки формируется надпись, содержащая текущее время:

```
<asp:Label ID="date" runat="server" Text="<%# DateTime.Now %>" />
```

Внутри конструкции `<%#  %>` можно вызывать методы страницы, а также методы и свойства компонентов страницы, лишь бы тип результирующего значения отвечал типу свойства, которому вы его присваиваете. Любое выражение связывания с данными вычисляется лишь после вызова метода `DataBind()`. Если метод `DataBind()` не вызывается, выражение не вычисляется.

При применении выражений связывания с данными внутри списковых и табличных элементов управления часто используется класс `DataBinder`. Он по-

зволяет генерировать выражения связывания с данными и осуществлять их разбор. В этом отношении особо важен его перегруженный статический метод `Eval()`. Данный метод, используя технологию отражения, выполняет синтаксический анализ выражения и вычисляет его значение. Синтаксис вызова метода `DataBinder.Eval()` обычно бывает таким:

```
<%# DataBinder.Eval(контейнер.DataItem, выражение) %>
```

Здесь опущен третий, необязательный, параметр - строка, содержащая установки форматирования возвращаемого значения. Выражение *контейнер*`.DataItem` возвращает объект, для которого вычисляется выражение. Обычно контейнером является текущий объект элемента управления, представляющий один отображаемый элемент данных.

Классический синтаксис вызова `DataBinder.Eval()` в ASP.NET 2.0 может быть несколько упрощен. Эквивалентом предыдущего выражения в ASP.NET 2.0 является выражение `<%# Eval(`*выражение*`) %>`.

ASP.NET 2.0 вводит новый тип выражений для связывания с данными – *динамические выражения*. В основе динамических выражений лежит новое семейство компонентов - *построители выражений*. Синтаксис динамических выражений подобен синтаксису выражений связывания с данными, но вместо символа `#` в качестве префикса в них используется символ `$`. Такие выражения анализируются при компиляции страницы. Выражение извлекается из её исходного кода, преобразуется в код на языке программирования и вставляется в класс страницы. Существует несколько предопределенных построителей выражений:

- `AppSettings:XXX` - возвращает значение заданной установки из раздела `<appSettings>` конфигурационного файла.
- `ConnectionStrings:XXX[.YYY]` - возвращает строку `XXX` из раздела `<connectionStrings>` конфигурационного файла. Необязательный параметр `YYY` позволяет указать, какой атрибут вас интересует - `connectionString` (по умолчанию) или `providerName`.
- `Resources:XXX,YYY` - возвращает значение глобального ресурса `YYY`, прочитанное из `.resx`-файла ресурсов `XXX`.

Точный синтаксис выражения определяется конкретным построителем, но в общем случае декларативное связывание свойств элементов управления с данными осуществляется следующим образом: `атрибут=<%$ выражение %>`.

5.14. СПИСКОВЫЕ ЭЛЕМЕНТЫ УПРАВЛЕНИЯ

Списковые элементы управления обеспечивают различные способы представления списков и таблиц. Название и назначение списковых элементов приведено в табл. 26.

Таблица 26

Списковые элементы управления

Элемент	Описание
BulletedList	Вывод элемента управления состоит из HTML-тэгов и , то есть представляет собой маркированный или нумерованный список ★
CheckBoxList	Создает HTML-элемент <table> или простой список, содержащий HTML-флажки
DropDownList	Создает на странице элемент <select> с атрибутом size="1", то есть раскрывающийся список с одной видимой строкой
ListBox	Создает элемент <select> с атрибутом size="n", чтобы построить обычное поле списка с единичным или множественным выбором и более чем одной видимой строкой
RadioButtonList	Создает HTML-элемент <table> или простой список, содержащий HTML-переключатели

Все списковые элементы управления имеют общего предка – класс ListControl. Полезные свойства данного класса описываются в табл. 27.

Таблица 27

Свойства класса ListControl

Имя свойства	Описание
AppendDataBoundItems	Булево значение; указывает, нужно ли добавлять загружаемые в элемент управления новые данные в конце существующего набора его данных ★
AutoPostBack	Булево значение; показывает, будет ли страница автоматически отправляться на сервер при изменении пользователем выбора в списке
DataMember	Имя таблицы в DataSource
DataSource	Источник данных для значений списка при заполнении списка путем связывания данных
DataTextField	Имя поля в DataSource, содержимое которого будет отображаемым текстом элементов списка
DataTextFormatString	Строка форматирования для значений из DataTextField (например, {0:C} для денежных сумм)
DataValueField	Имя поля в DataSource, содержимое которого будет значением элементов списка (свойство Value объекта ListItem)
Items	Коллекция элементов ListItem, содержащихся в списке
SelectedIndex	Индекс первого выбранного элемента в списке[1]
SelectedItem	Ссылка на первый выбранный элемент ListItem
SelectedValue	Значение первого выбранного элемента ListItem. Если у элемента задано свойство Value, то возвращается именно оно. Иначе возвращается значение свойства ListItem.Text

Кроме описанных свойств класс ListControl предоставляет серверное событие SelectedIndexChanged. Оно генерируется, когда выбор в списке изменяется и страница пересылается на сервер.

[1] Для того чтобы установить или извлечь несколько выбранных элементов, используется булево свойство Selected отдельного объекта ListItem.

Отдельные элементы списка данных хранятся в коллекции Items и представлены объектами класса ListItem. Свойство Text этого класса содержит выводимый текст элемента, свойство Value задает ассоциируемое с элементом значение, а булево свойство Selected указывает, выбран ли элемент. Статический метод ListItem.FromString() возвращает объект ListItem, созданный на основе строки-параметра.

Коллекцию Items можно заполнить посредством связывания данных или декларативно, используя набор тэгов <asp:ListItem>, размещаемых в контейнере спискового элемента управления. Ниже показан пример описания на странице элемента BulletedList.

```
<asp:BulletedList ID="myList" runat="server" BulletStyle="Square">
    <asp:ListItem Value="1" Text="One" />
    <asp:ListItem Value="2" Text="Two" />
    <asp:ListItem Value="3" Text="Three" />
</asp:BulletedList>
```

Каждый списковый элемент добавляет к своему базовому классу некоторые специфичные свойства и методы. Элементы управления CheckBoxList и RadioButtonList содержат свойства CellPadding - расстояние в пикселях между рамкой и ячейкой; CellSpacing - расстояние в пикселях между ячейками; RepeatColumns – количество выводимых столбцов; RepeatDirection – вывод списка по горизонтали или вертикали; RepeatLayout - значение, определяющее способ вывода элемента управления (Table (в виде таблицы) или Flow (просто группа элементов)), TextAlign – выравнивание текста. У класса ListBox имеется свойство Rows, содержащее число видимых строк элемента, и SelectionMode, определяющее, допустим ли множественный выбор. Элемент управления BulletedList позволяет настроить стиль маркера или номера (свойство BulletStyle), указать путь к изображению, которое будет использоваться в качестве маркера (свойство BulletImageUrl), и настроить значение, с которого будет начинаться нумерация (свойство FirstBulletNumber).

5.15. ЭЛЕМЕНТЫ УПРАВЛЕНИЯ ДЛЯ ИСТОЧНИКОВ ДАННЫХ

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.

5.16. ТАБЛИЦЫ, СВЯЗАННЫЕ С ДАННЫМИ

ASP.NET предлагает несколько элементов управления, предназначенных для табличного отображения данных: `Repeater`, `DataList`, `DataGrid`, `GridView` и `ListView`. Как основной элемент управления позиционируется `GridView`, появившийся в ASP.NET 2.0 и пришедший на смену `DataGrid`. Именно этот элемент управления будет рассмотрен подробно.

Элемент управления `GridView` был разработан с целью обеспечения автоматического двунаправленного связывания с применением минимума пользовательского кода. Он тесно связан с семейством новых компонентов, представляющих источники данных, и способен непосредственно обновлять их данные, лишь бы сами компоненты поддерживали такую возможность.

Для задания отображаемых столбцов элемент управления `GridView` имеет свойство-коллекцию `Columns`[1], которое можно определять как декларативно, так и программно. В последнем случае создаются необходимые объекты классов, производных от класса `DataControlField`, которые затем добавляются в коллекцию. Например, следующий код включает в состав элемента управления `gv` столбец, связанный с данными:

```
var field = new BoundField
            {
                DataField = "name",
                HeaderText = "Company Name"
            };
gv.Columns.Add(field);
```

1 У `GridView` есть булево свойство `AutoGenerateColumns`. Оно позволяет указать, должны ли столбцы автоматически создаваться для каждого поля источника данных. По умолчанию свойство имеет значение `true`, но автогенерация столбцов затрудняет их настройку.

Для объявления столбца в файле .aspx используется тэг <Columns>:

```
<Columns>
  <asp:BoundField DataField="customerid" HeaderText="ID" />
  <asp:BoundField DataField="name" HeaderText="Name" />
</Columns>
```

Столбцы данных выводятся в том порядке, в каком их объекты добавлялись в коллекцию. В табл. 28 перечислены производные от DataControlField классы, представляющие разные типы столбцов.

Таблица 28

Классы, представляющие разные типы столбцов

Класс	Описание
BoundField	Значение выводится в виде чистого текста (тип столбца по умолчанию)
ButtonField	Значение выводится в виде командной кнопки ссылочного или обычного кнопочного типа
CheckBoxField	Значение выводится в виде флажка; такие столбцы обычно применяются для представления значений булева типа
CommandField	Подобен классу ButtonField; представляет командную кнопку типа Select, Delete, Insert или Update
HyperLinkField	Представляет значение поля в виде гиперссылки; по щелчку на ней браузер выполняет переход по заданному URL
ImageField	Значение поля выводится как атрибут Src HTML-тэга <img>. Связанное поле должно содержать URL изображения
TemplateField	В ячейках столбца выводится пользовательский контент. Данный тип столбца используется в случаях, когда требуется создать пользовательский столбец. Шаблон, который вы связываете со столбцом, может содержать любое сочетание полей, связанных с данными, литералов, изображений и прочих элементов управления

Поле источника данных, выводимое в виде чистого текста, представляет класс BoundField. Чтобы указать, с каким полем связан объект этого класса, нужно присвоить имя поля свойству DataField. При желании можно задать для поля строку форматирования, присвоив её свойству DataFormatString. В свойстве NullDisplayText задается альтернативный текст, выводимый, когда поле содержит значение null. Установив свойство ConvertEmptyStringToNull в true, вы даете объекту BoundField указание трактовать пустые строки как значение null. Для вывода надписи в разделе верхнего или нижнего колонтитула, присвойте соответствующий текст свойству HeaderText или FooterText. Вместо текста в верхнем колонтитуле можно вывести изображение, задав его URL в свойстве HeaderImageUrl.

Кнопочное поле используется для размещения в столбце таблицы активных элементов. Обычно с помощью таких кнопок инициируются действия над текущей строкой. По щелчку кнопки выполняется возврат формы и генерируется событие RowCommand. Если таблица содержит несколько кнопочных столбцов, узнать, какую именно кнопку щелкнул пользователь, можно из свойства

CommandName - строка, уникально идентифицирующая кнопку в рамках элемента управления. Вот пример обработчика события RowCommand:

```
private void GV1_RowCommand(object sender,
                            GridViewCommandEventArgs e)
{
    if (e.CommandName.Equals("Add"))
    {
        // индекс строки, в которой щелкнул пользователь
        var index = Convert.ToInt32(e.CommandArgument);
        // добавляем товар в корзину для покупок
        AddToShoppingCart(index);
    }
}
```

Все кнопки в кнопочном столбце обычно имеют одну и ту же надпись; ее текст присваивается свойству Text объекта ButtonField. Если же вы хотите связать текст кнопок с определенным полем текущей строки данных, задайте имя этого поля в свойстве DataTextField.

Поле гиперссылки HyperLinkField позволяет пользователю перейти по другому URL. Этот URL можно непосредственно связать с полем источника данных. Имя поля в таком случае задается в свойстве DataNavigateUrlFields объекта HyperLinkField. Если же поля с URL в источнике данных нет, параметризованный URL кодируют в свойстве DataNavigateUrlFormatString.

```
<asp:HyperLinkField DataTextField="name" HeaderText="Product"
                    DataNavigateUrlFields="productid"
                    DataNavigateUrlFormatString="info.aspx?id={0}"
                    Target="ProductView" />
```

Если URL содержит более одного параметра, задайте в свойстве DataNavigateUrlFields разделенный запятыми список полей, значения которых будут подставляться на места параметров.

Столбец типа CheckBoxField связан с данными булевого типа, которые в нем представлены в виде установленных или снятых флажков. Булевы значения MS SQL Server хранит в столбцах типа Bit (в других СУБД - в столбцах аналогичных типов). Если же столбец связан с пользовательской коллекцией, то источником данных для столбца типа CheckBoxField может быть свойство типа bool. Попытка связать его с данными других типов приведет к исключению.

В столбце типа ImageField выводятся изображения. Ячейка такого столбца содержит тэг <img>, в атрибут Src которого должен подставляться URL изображения. Этот URL может содержаться в поле источника данных, заданном в свойстве DataImageUrlField, или составляться динамически с использованием параметра. Во втором случае параметризованный URL задается в свойстве DataImageUrlFormatString. Альтернативный текст, выводимый при отсутствии

изображения, также можно связывать с данными, для чего используется свойство DataAlternateTextField.

Столбец типа TemplateField позволяет задать шаблон для формирования значений столбца. Вы можете создать отдельные шаблоны для отображения значений в чётных и нечётных строках, для редактирования значений, а также для заголовка и нижнего колонтитула столбца. Свойства, представляющие эти шаблоны, перечислены в табл. 29.

Таблица 29

Свойства, представляющие шаблоны

Свойство	Описание шаблона
AlternatingItemTemplate	Определяет содержимое ячейки четной строки и способ его представления; если этот шаблон не задан, используется шаблон ItemTemplate
EditItemTemptate	Определяет содержимое ячейки редактируемой строки и способ его представления; этот шаблон должен содержать поля ввода и, возможно, проверочные элементы
FooterTemplate	Определяет содержимое нижнего колонтитула столбца
HeaderTemplate	Определяет содержимое заголовка столбца
ItemTemplate	Определяет содержимое ячейки столбца по умолчанию и способ его представления

В шаблон разрешается включать любые элементы: серверные элементы управления, литералы, выражения связывания с данными, используемые для получения значений полей текущей строки данных. Однако нужно учитывать, что выражения связывания с данными поддерживаются не для всех шаблонов, а только для тех, которые связаны со строками данных. Если попытаться использовать такое выражение в заголовке или нижнем колонтитуле, будет сгенерировано исключение.

Ниже приведен пример определения шаблона для столбца, содержащего информацию о товаре. Ячейка такого столбца содержит две строки - с названием товара и информацией о его расфасовке.

```
<asp:TemplateField HeaderText="Product">
    <ItemTemplate>
        <b><%# Eval("productname")%></b>
        <br /> available in <%# Eval("quantityperunit")%>
    </ItemTemplate>
</asp:TemplateField>
```

Элемент управления GridView поддерживает постраничный вывод, редактирование и сортировку информации в том случае, если это умеет делать связанный с ним компонент-источник данных. Источник данных уведомляет GridView о поддержке тех или иных возможностей посредством булевых свойств наподобие CanSort. Если же элемент управления GridView связан с данными через свойство DataSource, то GridView просто генерирует определен-

ные события, ожидая, что все необходимые действия будут выполнены пользовательским кодом их обработки.

Рассмотрим постраничный вывод информации в GridView. Если источником данных является компонент SqlDataSource, то его свойство DataSourceMode должно быть установлено в DataSet (значение по умолчанию). Это означает, что при каждом возврате формы из БД извлекается полный набор данных, а на странице при этом выводится немного записей. Если GridView связан с компонентом ObjectDataSource, то бизнес-объект, который служит источником данных, должен иметь встроенные функции разбиения на страницы. Вы конфигурируете ObjectDataSource таким образом, чтобы он правильно вызывал метод бизнес-объект, отвечающий за выдачу данных: задаете в свойствах StartRowIndexParameterName и MaximumRowsParameterName имена параметров, в которых этому методу передается начальный индекс и размер страницы. Также необходимо установить свойство ObjectDataSource.EnablePaging в значение true.

Когда свойство GridView.AllowPaging установлено в true, элемент управления выводит блок листания. Его конфигурация задается с помощью тэгов <PagerSettings> и <PagerStyle> или соответствующих им свойств. Страничный блок может содержать кнопки перехода к первой и последней страницам, а также изображения вместо обычных ссылок. Блок листания может работать в двух основных режимах: номеров страниц или кнопок относительной навигации. Режим работы блока листания задается в его свойстве Mode.

Для включения функции сортировки данных нужно установить свойство AllowSorting элемента управления GridView в true. Тогда щелчок заголовка столбца будет восприниматься, как команда отсортировать данные по значениям этого столбца. При желании можно связать со столбцом выражение сортировки, задав его в свойстве SortExpression. Такое выражение представляет собой разделенную запятыми последовательность имен столбцов. Каждое имя столбца может сопровождаться ключевым словом DESC или ASC, определяющим порядок сортировки (по убыванию и возрастанию соответственно). По умолчанию сортировка выполняется по возрастанию значений.

Как и в случае с листанием, реализация сортировки определяется объектом, представляющим источник данных. Если GridView связан с компонентом SqlDataSource, то данные извлекаются из базы в виде DataSet, формируется объект DataView, после чего вызывает его метод Sort(). Для получения от сервера БД заранее отсортированных данных нужно выполнить такие шаги:

1. Присвоить свойству DataSourceMode компонента SqlDataSource значение DataReader.
2. Написать хранимую процедуру, с помощью которой будут извлекаться данные. Очевидно, что хранимая процедура должна динамически составлять текст команды с предложением ORDER BY.

3. Присвоить свойству `SortParameterName` компонента `SqlDataSource` имя параметра хранимой процедуры, в котором будет передаваться выражение сортировки.

Учтите, что сортировка данных в базе несовместима с кэшированием. Поэтому установите свойство `SqlDataSource.EnableCaching` в `false`, иначе будет сгенерировано исключение. Как результат - при каждом возврате формы будет производиться обращение к базе данных.

Если `GridView` связан с компонентом `ObjectDataSource`, то задача сортировки перекладывается на слой доступа к данным, а с него - на СУБД. Свойству `SortParameterName` компонента `ObjectDataSource` нужно присвоить имя того параметра указанного метода, в котором задается выражение сортировки.

Операции листания и сортировки требуют выполнения возврата формы с последующим полным обновлением страницы. Однако благодаря введенной в ASP.NET 2.0 технологии *обратного вызова сценария* элемент управления `GridView` может сам обратиться к серверу, получить новый набор данных и обновить свой интерфейс, не вынуждая к обновлению всю страницу. Чтобы воспользоваться технологией, нужно установить булево свойство `EnableSortingAndPagingCallbacks` элемента управления `GridView` в `true`.

Кроме поддержки листания и сортировки, `GridView` позволяет редактировать свои данные прямо в таблице. Для включения поддержки редактирования нужно установить свойство `AutoGenerateEditButton` в `true`. Тогда в `GridView` автоматически появится дополнительный столбец с кнопками для перевода строк в режим редактирования. Свойство `EditIndex` в режиме редактирования содержит индекс редактируемой строки, а если оно имеет значение `-1`, то это говорит о том, что ни одна строка в данный момент не редактируется.

Коротко упомянем о возможностях других элементов управления для представления данных. Элемент управления `ListView` был представлен в ASP.NET 3.5. По своим функциям он схож с `GridView`, однако его настройка целиком базируется на шаблонах. `ListView` даёт возможность настроить при помощи шаблонов раскладку, строку данных, области для редактирования и другие части. `DataGrid` – это устаревший аналог `GridView`, который не обладает возможностью связывания с компонентом-источником данных. Если пытаться реализовать в `DataGrid` возможности листания и сортировки, то все это надо делать исключительно программно. `DataGrid` поддерживает набор колонок, но также как и `Repeater` и `DataList` может строить свой вывод на основе шаблонов. *Шаблон* (*template*) описывает отдельную строку табличного элемента, позволяя настраивать внешний вид этой строки. При выводе шаблон повторяется требуемое число раз, в зависимости от количества строк в источнике данных. В табл. 30 приведены допустимые шаблоны и указаны поддерживающие их элементы управления.

Таблица 30

Описание шаблонов

Шаблон	Описание	DataGrid	DataList	Repeater
ItemTemplate	Внешний вид элемента данных	Да	Да	Да
AlternatingItemTemplate	Если задан, то чередуется с ItemTemplate	Нет	Да	Да
HeaderTemplate	Вид заголовка	Да	Да	Да
FooterTemplate	Генерирует внешний вид колонтитула столбца	Да	Да	Да
SeparatorTemplate	Вид разделителя элементов данных	Нет	Да	Да
EditItemTemplate	Вид редактируемого элемента данных	Да	Да	Нет

5.17. ОТОБРАЖЕНИЕ ОТДЕЛЬНЫХ ЗАПИСЕЙ

В ASP.NET 2.0 появились два новых элемента управления, предназначенных для вывода информации из источника данных не в виде строк таблицы, а в виде отдельных записей. Это элементы управления DetailsView и FormView.

DetailsView - это связанный с данными элемент управления, выводящий на странице одну запись и необязательные кнопки для перехода между записями. Он обычно используется для обновления и вставки записей в сценариях с главным и подчиненным представлениями. Элемент управления DetailsView может быть связан с любым компонентом, представляющим источник данных. Он способен обновлять, удалять и вставлять записи в источнике данных, поддерживающем эти операции. В большинстве случаев для их выполнения не требуется писать ни строчки кода. Для настройки интерфейса элемента управления DetailsView достаточно выбрать необходимые поля данных и стили подобно тому, как это делается с элементом GridView.

Хотя элемент управления DetailsView обычно используется для модификации и вставки записей, он не проверяет, соответствуют ли введенные данные схеме их источника. Не предоставляет он и элементов пользовательского интерфейса, определяемых схемой данных (например, раскрывающегося списка для выбора значений внешнего ключа или специализированных шаблонов для редактирования данных определенных типов).

Организовать просмотр записей с помощью DetailsView исключительно просто. Вы размещаете этот элемент в форме, связываете его с компонентом, представляющим источник данных, и добавляете несколько декларативных установок. Ниже демонстрируется минимальная конфигурация элемента:

```
<asp:DetailsView ID="detailsView" runat="server"
                 DataSourceID="SqlDS" HeaderText="Performers">
</asp:DetailsView>
```

По умолчанию в элементе управления DetailsView выводятся все поля записи, с которой он связан, и соответствующие им вложенные элементы управ-

ления генерируются автоматически. Если же свойство AutoGenerateRows установить в false, то будут выводиться лишь те поля, которые явно перечислены в коллекции Fields. Элемент управления DetailsView поддерживает тот же набор типов полей, что и GridView.

На тот случай, если источник данных окажется пустым, можно задать шаблон EmptyDataTemplate, чтобы вывод элемента управления был более дружественным пользователю:

```
<asp:DetailsView ID="detailsView" runat="server"
                 DataSourceID="SqlDS" HeaderText="Performers">
    <EmptyDataTemplate>
        <asp:Label ID="lblEmpty" runat="server">
            There's no data to show in this view.
        </asp:Label>
    </EmptyDataTemplate>
</asp:DetailsView>
```

Когда свойство AllowPaging установлено в true, элемент DetailsView выводит блок листания, позволяющий пользователю переходить от записи к записи. Механизм листания элемента управления DetailsView использует свойство PageIndex, в котором содержится индекс текущей записи источника данных. Когда пользователь щелкает ту или иную навигационную кнопку, значение свойства PageIndex изменяется, элемент управления повторяет операцию связывания с данными и обновляет свой пользовательский интерфейс. Общее число записей в источнике данных возвращает свойство PageCount. Переход к другой записи сопровождается парой событий: PageIndexChanging и PageIndexChanged.

Обычно листание реализуется путем обработки серверных событий и требует полного обновления страницы. Однако это не единственная возможность: DetailsView обладает свойством EnablePagingCallbacks, позволяющим включить режим листания с использованием клиентской функции обратного вызова. Когда применяется технология обратного вызова сценария, возврат формы на сервер не производится, а данные новой записи обновляются путем асинхронного вызова сервера.

Элементы управления, подобные DetailsView, особенно полезны в тех случаях, когда основной работой пользователя является обновление, удаление и добавление данных. На командной панели элемента управления DetailsView имеются все необходимые для этого кнопки. Вы даете элементу указание создать эти кнопки, устанавливая в true свойства AutoGenerateEditButton, AutoGenerateDeleteButton и AutoGenerateInsertButton. Как и в случае с элементом управления GridView, все операции с данными DetailsView осуществляет связанный с ним компонент, представляющий источник данных. Для этого компонента нужно определить необходимые команды, а также задать в свойстве DataKeyNames ключ, идентифицирующий текущую запись.

Элемент управления DetailsView не поддерживает шаблонов редактирования и вставки записи, с помощью которых можно было бы полностью изменять

его пользовательский интерфейс. Вы можете задать набор шаблонов для отдельного поля, добавив это поле в элемент управления в виде объекта класса `TemplateField`:

```
<asp:TemplateField HeaderText="Country">
    <ItemTemplate>
        <asp:Literal runat="server" Text='<%# Eval("country")%>'/>
    </ItemTemplate>
    <EditItemTemplate>
        <asp:DropDownList ID="ddlCountries" runat="server"
                          DataSourceID="CountriesDS"
                          SelectedValue='<%# Bind("country")%>'/>
    </EditItemTemplate>
</asp:TemplateField>
```

Согласно этому определению содержимое поля `Country` в режиме просмотра выводится в литеральной форме, а в режиме редактирования - в виде раскрывающегося списка. Оператор `Bind()`, с помощью которого получается выделенное в списке значение, подобен оператору `Eval()`, но отличается от него двусторонним действием - он не только возвращает прочитанные из источника данные, но и записывает измененное значение в источник данных.

Элемент управления `FormView` можно считать расширенной версией элемента управления `DetailsView`. В отличие от элемента управления `DetailsView`, `FormView` не использует поля источника данных и требует, чтобы разработчик определял все поля с помощью шаблонов. Элемент управления `FormView` поддерживает все базовые операции с данными, разумеется, при условии, что их поддерживает и источник данных.

Поскольку пользовательский интерфейс элемента управления `FormView` определяется автором страницы, не приходится ожидать, что элемент сам будет реагировать на щелчок определенной кнопки и действовать соответственно. У него имеется несколько открытых методов (табл. 31), с помощью которых инициируются различные операции.

Таблица 31

Методы элемента управления `FormView`

Имя метода	Описание
`ChangeMode()`	Изменяет рабочий режим элемента управления; этому методу передается значение из перечисления `FormViewMode`, определяющее целевой режим: `Readonly`, `Edit` или `Insert`
`DeleteItem()`	Удаляет текущую запись элемента управления из источника данных
`InsertItem()`	Вставляет текущую запись элемента управления в источник данных. В момент вызова этого метода элемент управления должен находиться в режиме вставки, в противном случае будет выброшено исключение
`UpdateItem()`	Обновляет текущую запись элемента управления в источнике данных. В момент вызова этого метода элемент управления должен находиться в режиме редактирования, в противном случае будет выброшено исключение

Методы `InsertItem()` и `UpdateItem()` предназначены для инициирования соответствующих операций и вызываются элементами управления текущего шаблона. Им не нужно передавать ни вставляемой записи, ни новых значений, ни ключа удаляемой записи. Элемент управления `FormView` сам отлично знает, как извлечь необходимую информацию (он делает это так же, как элемент управления `DetailsView`). Методам `InsertItem()` и `UpdateItem()` передаётся значение булева типа, указывающее, должна ли производиться проверка данных. Если оно имеет значение `true`, активизируются проверочные элементы управления, определенные в составе шаблона.

5.18. КОНФИГУРИРОВАНИЕ ВЕБ-ПРИЛОЖЕНИЙ

Конфигурационная информация в ASP.NET имеет иерархическую структуру. Одна её часть хранится в файле `machine.config`, расположенном на верхнем уровне иерархии и содержащем информацию, относящуюся к компьютеру в целом, другая - в файлах `web.config`, которые служат узлами иерархического дерева. Заданные в них установки в зависимости от местонахождения этих файлов относятся ко всем установленным на данном компьютере приложениям ASP.NET, отдельному приложению или группе его страниц, хранящихся в одном каталоге. Параметры конфигурации ASP.NET обычно задаются в секции `<system.web>` файла конфигурации. Некоторые подразделы этой секции перечислены в табл. 32

Таблица 32

Некоторые подразделы `<system.web>`

Имя раздела	Описание
`<authentication>`	Настройки механизма аутентификации
`<authorization>`	Список авторизированных пользователей
`<customErrors>`	Установки пользовательских страниц с сообщениями об ошибках
`<globalization>`	Параметры локализации приложения
`<httpCookies>`	Свойства `cookie`, используемых приложением ASP.NET ★
`<httpHandlers>`	Список зарегистрированных обработчиков HTTP
`<httpModules>`	Список зарегистрированных модулей HTTP
`<httpRuntime>`	Параметры исполняющей среды HTTP
`<identity>`	Включение поддержки имперсонализации
`<machineKey>`	Ключ шифрования для данных, требующих защиты
`<membership>`	Параметры аутентификации пользователей через членство в ASP.NET ★
`<pages>`	Функции страниц ASP.NET
`<profile>`	Параметры модели данных пользовательского профиля ★
`<roleManager>`	Параметры управления ролями ★
`<sessionState>`	Конфигурация объекта `Session`
`<siteMap>`	Параметры поддержки навигационной инфраструктуры ★
`<trace>`	Конфигурация системы трассировки

Отметим, что файл конфигурации может иметь специальный раздел `<location>`. С его помощью назначают индивидуальные установки подкаталогам приложения. У раздела `<location>` два атрибута: `path` и `allowOverride`. Ат-

рибут `path` представляет виртуальный путь, к которому применяются установки данного раздела. В следующем примере показано, как он действует. Обратите внимание: имя папки должно быть относительным и не должно начинаться с прямой или обратной косой черты либо точки.

```
<configuration>
  <system.web>
    <!-- Здесь располагаются установки для приложения -->
  </system.web>
  <location path="Reserved">
    <system.web>
      <!-- Здесь --- установки для подпапки /Reserved -->
    </system.web>
  </location>
</configuration>
```

Таким образом, использование `<location>` позволяет иметь единственный файл `web.config`, обеспечивающий централизованное хранение установок, но при этом можно конфигурировать каждый подкаталог отдельно.

В составе ASP.NET имеется мощный API для управления конфигурациями с функциями для чтения, записи и навигации по конфигурационным файлам приложения. Специальное пространство имен `System.Configuration` отвечает за работу с файлами конфигураций. Наборы конфигурационных установок представлены в программе строго типизированными объектами. Например, класс `WebConfigurationManager` предназначен для работы с установками из приложений ASP.NET.

5.19. ИНФРАСТРУКТУРА ОБРАБОТКИ ЗАПРОСА

Рассмотрим стадии обработки клиентских запросов компонентами ASP.NET. Итак, пусть клиент направил запрос, URI которого определяет его как относящийся к ASP.NET. На сервере выполнятся следующие шаги.

1. Веб-сервер IIS на основании расширения URI принимает решение о вызове библиотеки `aspnet_isapi.dll`, написанной на обычном (не на управляемом) коде, и передаёт этой библиотеке данные запроса для обработки.
2. `aspnet_isapi.dll` запускает (при необходимости) рабочий процесс `aspnet_wp.exe`, написанный на управляемом коде.
3. В рамках `aspnet_wp.exe` вызывается статический метод `HttpRuntime.ProcessRequest()`, который становится ответственным за обработку запроса. В качестве параметра данный метод принимает объект класса `HttpWorkerRequest`, совмещающий как данные запроса, так и последующие данные ответа.
4. Порождается объект класса `HttpContext`, содержащий отдельные объекты для представления запроса, ответа, сессии и прочие. Подробнее класс `HttpContext` будет рассмотрен ниже.

5. Метод `HttpApplication.GetApplicationInstance()` порождает или возвращает существующий домен для веб-приложения. Дальнейшая работа происходит в рамках этого домена.
6. Срабатывает цепочка HTTP-модулей, являющихся перехватчиками отдельных событий запроса и (при сформированном ответе) ответа.
7. Порождается фабрикой классов или выбирается из списка зарегистрированный HTTP-обработчик. Его задача – обработать запрос в зависимости от конкретного расширения URI.
8. Данные, сформированные обработчиком, передаются в качестве ответа на сервер, отправляющий их клиенту.

Важную роль в процессе обработки HTTP-запроса играет объект класса `HttpContext`. Он выступает в роли своеобразного «клея», объединяющего все стадии запроса. Табл. 33 содержит описание свойств класса `HttpContext`.

Таблица 33

Свойства класса `HttpContext`

Свойство	Описание
`AllErrors`	Массив объектов `Exception`, каждый из которых представляет ошибку, происходящую при выполнении запроса
`Application`	Экземпляр класса `HttpApplicationState`, содержащий глобальное совместно используемое состояние приложения
`ApplicationInstance`	Объект `HttpApplication` для текущего запроса, точнее, объект класса, определенного в файле `global.asax`
`Cache`	Связанный с текущим запросом объект `Cache`
`Current`	Статическое свойство - текущий объект `HttpContext`
`CurrentHandler`	Возвращает обработчик текущего запроса ★
`Error`	Возвращает первое исключение, сгенерированное при обработке текущего запроса
`Handler`	Возвращает и позволяет задать для текущего запроса обработчик HTTP
`IsCustomErrorEnabled`	Указывает, включена ли для текущего запроса пользовательская обработка ошибок
`IsDebuggingEnabled`	Указывает, выполняется ли запрос в режиме отладки
`Items`	Хэш-таблица, которая для модулей и обработчиков HTTP может служить средством совместного доступа к данным
`PreviousHandler`	Возвращает объект `IHttpHandler` родительского обработчика
`Profile`	Объект, представляющий профиль текущего пользователя ★
`Request`	Экземпляр класса `HttpRequest`, представляющий текущий запрос HTTP
`Response`	Экземпляр класса `HttpResponse`, отправляющий клиенту данные ответа HTTP
`Server`	Экземпляр класса `HttpServerUtility`, предоставляющего вспомогательные методы для обработки запросов
`Session`	Экземпляр класса `HttpSessionState`, управляющий данными, связанными с конкретным сеансом

`SkipAuthorization`	Позволяет указать, должен ли модуль, осуществляющий авторизацию на основе URL, пропустить для текущего запроса этап авторизационной проверки. Свойство используется главным образом модулями аутентификации, которым нужно перенаправлять пользователя на страницу, с разрешённым анонимным доступом
`Timestamp`	Возвращает объект `DateTime`, представляющий начальный штамп времени текущего запроса
`Trace`	Объект `TraceContext` текущего запроса
`User`	Объект `IPrincipal`, представляющий пользователя, от которого поступил запрос

В ASP.NET программист может создавать собственные HTTP-обработчики, производящие непосредственную обработку запрашиваемых ресурсов с определенным расширением. Технически, HTTP-обработчик – это класс, реализующий интерфейс `System.Web.IHttpHandler` и зарегистрированный в инфраструктуре ASP.NET при помощи конфигурационных файлов.

В качестве примера рассмотрим создание обработчика, который будет перехватывать все запросы к страницам с расширением `*.calc` и выводить сумму двух чисел, переданных как параметры GET-запроса. Первое, что необходимо выполнить – написать класс с реализацией `IHttpHandler`. Интерфейс `IHttpHandler` содержит следующие элементы:

- `void ProcessRequest(HttpContext context)` – в этом методе происходит обработка запроса и формирование ответа с использованием данных объекта `context`;
- `bool IsReusable { get; }` – это свойство возвращает `true`, если один объект-обработчик можно использовать для нескольких запросов.

Наш обработчик будет реализован как класс `CalcHandler`, размещенный в сборке `UserHandlers.dll`.

```
// CalcUserHandler.cs file
using System;
using System.Web;

namespace UserHandlers
{
    public class CalcHandler : IHttpHandler
    {
        public void ProcessRequest(HttpContext context)
        {
            var objRequest = context.Request;
            var result = Int32.Parse(objRequest.Params["p1"]) +
                         Int32.Parse(objRequest.Params["p2"]);
            var objResponse = context.Response;
            objResponse.Write("<html><body><h1>");
            objResponse.Write(result.ToString());
            objResponse.Write("</h1></body></html>");
        }
```

```
        public bool IsReusable
        {
            get { return true; }
        }
    }
}
```

Скомпилированную сборку `UserHandlers.dll` разместим в каталоге `Bin` некоего веб-приложения (в случае создания глобальных обработчиков следует использовать GAC).

Теперь зарегистрируем обработчик. Для этого используется секция `<httpHandlers>` в конфигурационных файлах. Секция может содержать элементы `<add>`, `<remove>` и `<clear>`. Атрибутами `<add>` являются:

`verb` – специфицирует глагол протокола HTTP. Можно указать несколько глаголов через запятую (`GET, POST`) либо использовать символ `*`;

`path` – отдельный файл или шаблон файлов (расширение), к которым применяется обработчик;

`type` – имя класса обработчика и имя сборки, в которой он находится.

Для регистрации нашего обработчика используется такой файл `web.config`:

```
<configuration>
  <system.web>
    <httpHandlers>
      <add verb="*" path="*.calc"
           type="UserHandlers.CalcHandler, UserHandlers"/>
    </httpHandlers>
  </system.web>
</configuration>
```

Кроме регистрации в конфигурационном файле, расширение `.calc` должно быть зарегистрировано в IIS, чтобы вызывалась `aspnet_isapi.dll`.

Альтернативный способ определения обработчика HTTP предполагает использование файла `.ashx`. Такой обработчик активизируется при получении запроса ресурса `.ashx`. Связь между обработчиком и ресурсом `.ashx` устанавливается в самом файле ресурса с помощью специальной директивы `@WebHandler`:

```
<%@ WebHandler Language="C#" Class="Handler" %>
```

При получении HTTP-запроса с указанием URL, соответствующего ресурсу `.ashx`, автоматически вызывается класс `Handler`. Ниже приведен пример содержимого файла `.ashx`. Как видите, это просто файл класса, в который добавлена директива `@WebHandler`.

```
<%@ WebHandler Language="C#" Class="Handler" %>
using System;
using System.Web;

public class Handler : IHttpHandler
{
```

```
    public void ProcessRequest(HttpContext context)
    {
        context.Response.ContentType = "text/plain";
        context.Response.Write("Hello World");
    }

    public bool IsReusable
    {
        get { return false; }
    }
}
```

Когда для реализации обработчика HTTP используется ресурс `.ashx`, никакой дополнительной настройки для его инсталляции не требуется.

5.20. ВЕБ-ПРИЛОЖЕНИЕ И ФАЙЛ GLOBAL.ASAX

Каждое веб-приложение описывается объектом класса `HttpApplication`, или класса-наследника `HttpApplication`. Свойства данного класса описаны в табл. 34.

Таблица 34

Свойства класса `HttpApplication`

Имя свойства	Описание
`Application`	Объект класса `HttpApplicationState`, описывающий состояние веб-приложения
`Context`	Объект класса `HttpContex`, описывающий контекст запроса
`Modules`	Коллекция *модулей* – специальных обработчиков, дополняющих функции работы с запросом пользователя
`Request`	Ссылка на объект `HttpRequest`, обеспечивающий доступ к информации о HTTP-запросе
`Response`	Ссылка на объект `HttpResponse`, обеспечивающий доступ к информации о HTTP-ответе
`Server`	Объект класса `HttpServerUtility`, описывающий параметры веб-сервера
`Session`	Ссылка на объект класса `HttpSessionState`, хранящий данные текущей сессии пользователя в веб-приложении
`User`	Ссылка на объект, реализующий интерфейс `IPrincipal` и описывающий пользователя. Используется при аутентификации

Для разработчика важной является возможность перехватывать события приложения. В табл. 35 перечислены некоторые события приложения, предоставляемые классом `HttpApplication`. Большинство из них генерируются при обработке приложением каждого запроса.

Таблица 35

Некоторые события веб-приложения

Событие	Причина срабатывания	№
`BeginRequest`	Получение нового запроса	1
`AuthenticateRequest`	Завершение аутентификации пользователя	2

AuthorizeRequest	Завершение авторизации пользователя	3
ResolveRequestCache	Генерируется после авторизации, но перед запуском обработчика. Используется модулями кэширования для отмены выполнения обработчиков запроса, если в кэше есть нужная запись	4
AcquireRequestState	Загрузка состояния сеанса	5
PreRequestHandlerExecute	Перед передачей запроса обработчику	6
PostRequestHandlerExecute	Завершение обработчика запроса	7
ReleaseRequestState	После завершения всех обработчиков запроса. Используется модулями состояний для сохранения значений состояния	8
UpdateRequestCache	После завершения обработчика. Используется модулями кэширования для сохранения ответа в кэше	9
EndRequest	После обработки запроса	10
Disposed	Перед закрытием приложения	–
Error	При наступлении необработанной исключительной ситуации	–
PreSendRequestContent	Перед передачей клиенту содержимого ответа	–
PreSendRequestHeaders	Перед передачей клиенту заголовков HTTP	–

Для реализации перехватчика события можно создать пользовательский HTTP-модуль. Но простейшим вариантом является использование файла `global.asax`. Этот файл играет роль пункта реализации глобальных событий, объектов и переменных.

Файл `global.asax` поддерживает три директивы: `@Application`, `@Import`, `@Assembly`. Директива `@Application` позволяет определить базовый класс, на основе которого создается класс приложения (атрибут `Inherits`), указать язык программирования для серверного кода обработчиков событий (атрибут `Language`), а также задать описание приложения (атрибут `Description`).

Приведем пример файла `global.asax`, содержащего обработчики событий `BeginRequest` и `EndRequest` (заметим, что при использовании Visual Studio файл `global.asax` будет сопровождаться файлом Code Behind).

```
<%@ Application Language="C#" %>

<script runat="server">
    void Application_BeginRequest(object sender, EventArgs e)
    {
        Response.Write("Request starts!" + "<br />");
    }
    void Application_EndRequest(object sender, EventArgs e)
    {
        Response.Write("Request ends!" + "<br />");
    }
</script>
```

Некоторые события можно обработать, используя только обработчики, размещенные в файле `global.asax`. Это события `Application_Start` (запуск

приложения), Application_End (завершение приложения), Session_Start (начало сеанса пользователя), Session_End (завершение сеанса пользователя).

5.21. МОДЕЛЬ ПОСТАВЩИКОВ

Модель поставщиков, появившаяся в ASP.NET 2.0, даёт разработчикам возможность заменять встроенные компоненты ASP.NET, а также делать заменяемыми и настраиваемыми компоненты собственных приложений. Для стандартных функций ASP.NET, таких как, например, управление членством, состоянием или пользовательскими профилями, в этой системе имеется несколько готовых поставщиков, один из которых является используемым по умолчанию. Настраивая среду исполнения приложения ASP.NET при помощи конфигурационных файлов, можно выбрать для каждой функции системы тот поставщик, который лучше подходит для нужд данного приложения.

В ASP.NET 2.0 каждый поставщик представляет собой класс, наследующий некий базовый класс, но переопределяющий его поведение. Для создания собственного поставщика заданного сервиса вы определяете новый класс. Все поставщики поддерживаемых сервисов являются производными от одного базового класса ProviderBase. В табл. 36 перечислены базовые классы и соответствующие им функции поставщиков.

Таблица 36

Базовые классы поставщиков и соответствующие сервисы

Имя класса	Описание сервиса
MembershipProvider	Управление учетными записями пользователей
PersonalizationProvider	Управление персонализацией компонентов Web Parts
ProfileProvider	Сохранение информации пользовательских профилей
ProtectedConfigurationProvider	Шифрование информации в файлах конфигурации
RoleProvider	Управление информацией о ролях пользователей
SessionStateStoreProviderBase	Работа с данных состояния сеанса пользователя
SiteMapProvider	Управление информацией карты сайта
WebEventProvider	Мониторинг состояния системы и приложения, обработка системных событий

Рассмотрим некоторые поставщики. В ASP.NET имеются два поставщика членства: SqlMembershipProvider и ActiveDirectoryMembershipProvider. Они оба определены в пространстве имен System.WebSecurity. Функциями поставщика SqlMembershipProvider являются сохранение информации о членстве в специальных таблицах базы данных. Поставщик ActiveDirectoryMembershipProvider управляет хранением информации о членстве в Active Directory.

В ASP.NET имеются три реализации базового класса поставщика ролей: SqlRoleProvider, WindowsTokenRoleProvider и AuthorizationStoreRoleProvider. Класс SqlRoleProvider (используемый по умолчанию поставщик ролей) хранит связи между пользователями и ролями в базе данных. WindowsTokenRoleProvider – это поставщик ролей, использующий для получения сведений о пользователях информацию из системы групп безопасности

Windows. Он предназначен в первую очередь для интранет-приложений ASP.NET, в которых применяется аутентификация Windows, и запрещен анонимный доступ. Возвращаемая поставщиком информация о ролях пользователя основана на его членстве в определенной группе Windows. Данный поставщик нельзя использовать для создания и удаления ролей. Поставщик `AuthorizationStoreRoleProvider` управляет хранением информации о ролях, предназначенной для менеджера авторизации AzMan.

В ASP.NET имеется единственный встроенный поставщик профилей – `SqlProfileProvider`, определенный в пространстве имен `System.Web.Profile`. Для хранения данных профиля он использует таблицу базы данных. Обязанность поставщика профиля – считывать данные профиля из хранилища в начале выполнения запроса и записывать измененные значения обратно по завершении его выполнения. Таблица поставщика содержит по одной записи на каждого пользователя приложения. Пользовательские установки представлены в виде последовательности значений, в которой различаются текст и двоичные данные.

Управление состоянием сеанса – одна из основ ASP.NET. Для каждого сеанса работы пользователя с приложением система сохраняет коллекцию пар «имя=значение», определяющих состояние сеанса, и предоставляет API чтения и записи этих данных, реализованный в виде объекта `Session`. Поставщик состояния сеанса – это компонент, ответственный за предоставление данных, связанных с текущим сеансом. В ASP.NET имеются три предопределенных поставщика, хранящих данные в памяти рабочего процесса, на сервере состояния и в базе данных MS SQL Server. Используемый по умолчанию поставщик `InProcSessionStateStore` хранит данные в виде живых объектов в кэше ASP.NET. Поскольку эти данные доступны в любой момент, поставщик `InProcSessionStateStore` является наиболее быстрым среди своих собратьев. Однако чем больше данных состояния сеанса он хранит, тем больше потребляет памяти сервера, увеличивая тем самым риск снижения производительности. Поставщик состояния сеанса `OutOfProcSessionStateStore` хранит данные вне рабочего процесса ASP.NET. Если говорить конкретнее, он держит их в памяти сервиса Windows, имя которого – `aspnet_state.exe`. По умолчанию этот сервис остановлен, и его нужно запускать вручную. Поставщик `SqlSessionStateStore` хранит данные состояния сеанса в базе данных MS SQL Server. Вы можете хранить данные на любом компьютере, с которым у веб-сервера имеется связь и на котором выполняется MS SQL Server версии 7.0 или выше. По умолчанию поставщиком `SqlSessionStateStore` используется база данных `ASPState`, содержащая несколько хранимых процедур.

Стандартные поставщики ASP.NET 2.0, сохраняющие информацию в базе данных MS SQL Server, работают с определенным набором таблиц. Ниже указано назначение каждой таблицы.

Таблица 37

Описание таблиц стандартных поставщиков

Имя таблицы	Описание
aspnet_Applications	Перечень приложений, использующих базу данных
aspnet_Membership	Информация о членстве пользователей приложений; перечень пользователей хранится в таблице aspnet_Users
aspnet_Paths	Пути к страницам, использующим Web Parts
aspnet_PersonalizationAllUsers	Информация о структуре страниц приложений, использующих Web Parts, относящаяся ко всем пользователям
aspnet_PersonalizationPerUser	Относящаяся к конкретным пользователям информация о структуре страниц приложений, использующих Web Parts
aspnet_Profile	Данные профиля для каждого из пользователей
aspnet_Roles	Список всех доступных ролей
aspnet_SchemaVersions	Информация о версиях схемы таблиц, поддерживаемых каждой функцией
aspnet_Users	Перечень всех зарегистрированных пользователей. Данная таблица совместно используется всеми поставщиками, которым требуется информация о пользователях
aspnet_UsersInRoles	Информация о соответствии между пользователями и ролями
aspnet_WebEvent_Events	Сведения о запротоколированных событиях Web

Для создания описанных таблиц можно использовать утилиту aspnet_regsql.exe или инструмент Web Site Administration Tool (WSAT), доступный в Visual Studio 2008 через меню Website.

5.22. ПОДДЕРЖКА СОХРАНЕНИЯ СОСТОЯНИЯ

Специфика большинства веб-приложений подразумевает хранение информации (состояния) при переходе от одной страницы приложения к другой. Типичным примером является Internet-магазин, в котором пользователь просматривает страницы с информацией о товаре, помещает некоторые товары в свою корзину покупок, а затем оформляет окончательный заказ. Протокол HTTP не имеет стандартных средств для поддержки сохранения состояний, поэтому данная задача реализуется внешними по отношению к протоколу средствами. Платформа ASP.NET предоставляет программистам достаточный встроенный «арсенал» средств управления состояниями, что зачастую избавляет от написания лишнего кода.

ASP.NET поддерживает четыре основных типа состояний: *состояние приложения*, *состояние сеанса*, *cookie* и *состояние представления* (View State). Табл. 38 предназначена для характеристики и сравнения различных типов состояний.

Таблица 38

Характеристика различных типов состояний

Тип	Область видимости	Преимущества	Недостатки
Состояние приложения	Глобальная в приложении	Совместно используется всеми клиентами	Не могут вместе использоваться многими компьютерами. Область применения перекрывается средствами кэширования данных
Состояние сеанса	Для клиента	Может конфигурироваться на совместное использование многими компьютерами	Требуется коррекция файлов cookie и адресов URL для управления взаимодействием клиентов
Cookie	Для клиента	Не зависит от настроек сервера. Сохраняется у клиента. Может существовать после завершения сеанса	Ограниченный объем памяти (около 4 Кбайт). Клиент может отключить поддержку cookie. Состояние передаётся в обоих направлениях с каждым запросом
Состояние представления	На уровне запросов POST к этой же странице	Не зависит от настроек сервера	Сохраняется, только если запрос POST передается этой же странице. Передаётся в обоих направлениях с каждым запросом

К состоянию приложения можно обратиться, используя свойство `Application` класса страницы или приложения. Свойство возвращает объект типа `HttpApplicationState`. Этот класс является коллекцией, хранящей данные любых типов в парах ключ/значение, где ключом является строка или числовой индекс. Пример использования состояния приложения приведен в следующем фрагменте кода. Как только приложение запускается, оно загружает данные из БД. После этого для обращения к данным можно сослаться на кэшированную версию объекта:

```
// фрагмент файла global.asax:
protected void Application_Start(object src, EventArgs e)
{
    var ds = new DataSet();
    // здесь как-то заполняем набор данных (не показано)
    // кэшируем ссылку на набор данных
    Application["FooDataSet"] = ds;
}

// фрагмент файла страницы:
protected void Page_Load(object src, EventArgs e)
{
    myGridView.DataSource = (DataSet)(Application["FooDataSet"]);
}
```

Состояние приложения разделяется между всеми клиентами приложения. Это значит, что несколько клиентов одновременно могут осуществлять доступ (чтение или запись) к данным в состоянии приложения. Класс `HttpApplicationState` имеет встроенные средства контроля целостности данных. В некоторых ситуациях можно вручную вызвать методы класса `Lock()` и `Unlock()` для установки и снятия блокировки состояния приложения.

В большинстве веб-приложений требуется хранить некоторую информацию персонально для каждого клиента (например, для отслеживания заказанных товаров при посещении интернет-магазина). ASP.NET предполагает использование для этих целей состояний сеансов (или *сессий*). Когда с приложением начинает взаимодействовать новый клиент, новый *идентификатор сеанса* автоматически генерируется и ассоциируется с каждым последующим запросом этого же клиента (или с помощью cookies, или путем коррекции URL).

Состояния сеанса поддерживаются в экземпляре класса `HttpSessionState` и доступны через свойство `Session` страницы или объекта `HttpContext`. Табл. 39 показывает основные элементы класса `HttpSessionState`.

Таблица 39

Элементы класса `HttpSessionState`

Имя элемента	Описание
`Count`	Количество элементов в коллекции
`IsCookieless`	Булево свойство; указывает, используются ли cookie для сохранения идентификатора сессии
`IsNewSession`	Булево свойство; `true`, если сессия создана текущим запросом
`IsReadOnly`	Свойство равно `true`, если элементы сессии доступны только для чтения
`Mode`	Режим сохранения объекта сессии, одно из значений перечисления `SessionStateMode`: `InProc` – объект сохраняется в рабочем процессе ASP.NET, `Off` – отсутствует поддержка сессий, `StateServer` – за сохранение объекта сессии отвечает внешняя служба, `SQLServer` – объект сессии сохраняется в базе данных, `Custom` – за сохранение сессии отвечает пользовательский поставщик
`SessionID`	Строка с уникальным идентификатором сессии
`Timeout`	Время жизни сессии в минутах
`Add()`	Метод для добавления элемента в коллекцию сессии
`Abandon()`	Метод вызывает уничтожение сессии
`Clear()`	Удаляет все элементы коллекции, аналог `RemoveAll()`
`Remove()`	Удаляет элемент с указанным именем (ключом)
`RemoveAll()`	Удаляет все элементы коллекции
`RemoveAt()`	Удаляет элемент в указанной позиции

Для доступа к отдельным элементам, хранящимся в сессии, можно использовать строковый или целочисленный индексатор.

Конфигурирование состояния сеанса выполняется при помощи секции `<sessionState>`. Возможные атрибуты этой секции описаны в табл. 40. Большинство атрибутов введены в ASP.NET 2.0, а от ASP.NET 1.0 в неизменном виде унаследовано только четыре атрибута: `mode`, `timeout`, `stateConnectionString` и `sqlConnectionString`. Атрибут `cookieless` в ASP.NET 1.0 имел булев тип.

Таблица 40

Атрибуты раздела `<sessionState>`

Атрибут	Описание
`allowCustomSqlDatabase`	Если равен `true`, данные сеанса могут храниться в заданной вами таблице БД, а не в стандартной `ASPState`
`cookieless`	Определяет, как идентификатор сеанса должен передаваться клиенту
`cookieName`	Имя cookie, если для хранения идентификаторов сеанса используются cookie
`customProvider`	Имя пользовательского поставщика состояния сеанса
`mode`	Определяет, где должны храниться данные состояния
`regenerateExpiredSessionId`	Если равен `true`, при получении запроса с просроченным идентификатором сеанса генерируется новый идентификатор, иначе прежний идентификатор снова становится действительным. По умолчанию атрибут имеет значение `false`
`sessionIDManagerType`	Идентифицирует компонент, который будет использоваться в качестве генератора идентификаторов сеанса. По умолчанию данный атрибут имеет значение `null`
`sqlCommandTimeout`	Определяет, как долго SQL-команда может не возвращать результат, прежде чем она будет отменена. По умолчанию данный атрибут имеет значение 30 секунд
`sqlConnectionString`	Строка подключения к SQL Server
`stateConnectionString`	Имя сервера, на котором будет удаленно храниться состояние сеанса, или его адрес и порт
`stateNetworkTimeout`	Определяет, как долго сетевое соединение TCP/IP между Web-сервером и сервером состояния может отсутствовать, прежде чем запрос будет отменён. По умолчанию данный атрибут имеет значение 10 секунд
`timeout`	Определяет, как долго сеанс может быть бездействующим, прежде чем будет прекращен. По умолчанию данный атрибут имеет значение 20 минут
`useHostingIdentity`	Указывает, какая учетная запись должна использоваться при доступе к пользовательскому поставщику состояния или поставщику SQL Server в случае применения метода интегрированной защиты. Значение `true` (по умолчанию) соответствует использованию учетной записи хоста, то есть учетной записи ASP.NET в системе Windows. Если же задано значение `false`, используется учетная запись клиента (то есть процесс ASP.NET имперсонализирует учетную запись клиента)

Для того чтобы состояние сеанса сохранялось между запросами, клиент должен иметь возможность передать его идентификатор серверному приложению. Как это делается, определяет атрибут `cookieless`. Он может принимать одно из значений, которые относятся к перечислимому типу `HttpCookieMode`: `AutoDetect` - использовать cookie только при условии, что браузер их поддерживает, `UseCookies` - использовать cookie для сохранения идентификатора сеанса независимо от поддержки браузера, `UseDeviceProfile` - решение об исполь-

зовании cookie зависит от возможностей браузера, которые перечислены в разделе профилей устройств в конфигурационном файле, `UseUri` - передавать идентификатор сеанса в URL независимо от того, поддерживает ли браузер cookie.

Платформа ASP.NET предоставляет возможность хранить сессии вне рабочего процесса. Если атрибут `mode` элемента `sessionState` файла `web.config` установлен в `StateServer` или в `SqlServer`, то ASP.NET хранит сессии в другом процессе (выполняется как служба) или в базе данных MS SQL Server. Если используется сохранение сессии вне рабочего процесса, то любой тип, хранящийся в сессии, должен быть сериализуемым.

Пусть атрибут `mode` установлен в `StateServer`. В этом случае на компьютере, который будет ответственным за хранение сессий (это не обязательно должен быть компьютер, на котором выполняется рабочий процесс ASP.NET) должна быть запущена служба *ASP.NET state service*. Именно в адресном пространстве этой службы будут храниться данные. По умолчанию служба прослушивает порт 42424, но этот номер можно изменить при помощи ключа `HKLM\SYSTEM\CurrentControlSet\Services\aspnet_state\Parameters\Port`.

Еще один способ хранения состояний сеансов вне процесса – их размещение в базе данных SQL Server. Прежде чем задать этот режим, нужно выполнить сценарий `InstallSqlState.sql`[1]. Главная задача сценария – создать таблицу в базе `tempdb`, способную хранить сессии отдельных клиентов и индексированную идентификаторами сессий. В конфигурационном файле указывается сервер БД, пользователь и пароль.

По умолчанию ASP.NET предполагает, что каждая страница просит загрузить состояния сеанса во время инициализации страницы и сохранить состояние после вывода страницы. Это означает два обращения к серверу состояний или базе данных при хранении сессий вне рабочего процесса. Атрибут `EnableSessionState` директивы `@Page` позволяет более тонко настроить работу с сессиями. Если атрибут установлен в значение `ReadOnly`, то сессии загружаются, но не сохраняются после вывода страницы. Если же установить атрибут в `false`, то работа с сессиями на странице отключается.

Для управления данными cookie платформа ASP.NET предоставляет класс `HttpCookie`. Классы `HttpResponse` и `HttpRequest` содержат свойство-коллекцию `Cookies` типа `HttpCookieCollection` для работы с установленными cookie клиента. Определение класса `HttpCookie` представлено ниже:

```
public sealed class HttpCookie
{
    // Открытые конструкторы
    public HttpCookie(string name);
    public HttpCookie(string name, string value);

    // Открытые свойства
```

[1] Данный сценарий находится в главном каталоге .NET.

```
    public string Domain { set; get; }
    public DateTime Expires { set; get; }
    public bool HasKeys { get; }
    public string Name { set; get; }
    public string Path { set; get; }
    public bool Secure { set; get; }
    public string this[string key] { set; get; }
    public string Value { set; get; }
    public NameValueCollection Values { get; }
}
```

Каждый файл cookie может хранить много пар имя/значение, доступных посредством коллекции `Values` класса `HttpCookie` или посредством индексатора по умолчанию этого класса. Важно отметить следующее:

1. Объем информации, хранящейся в cookie, ограничен 4 Кбайтами.
2. И ключ, и значение сохраняются в cookie в виде строк.
3. Некоторые браузеры могут отключить поддержку cookie.

Чтобы вынудить клиента установить cookie, добавьте новый экземпляр класса `HttpCookie` в коллекцию `Cookies` перед выводом страницы. При этом можно проверить, поддерживает ли клиентский браузер cookie, при помощи булевого свойства `Request.Browser.Cookies`. Для доступа к файлу cookie, переданном клиентом в запросе, нужно обратиться к коллекции `Cookies`, являющейся свойством объекта запроса.

В следующем листинге представлена страница, устанавливающая и использующая cookie с именем `name`:

```
<%@ Page Language="C#" %>

<script runat="server">
protected void Page_Load(object sender, EventArgs e)
{
    if(Request.Cookies["name"] != null)
    {
        lbl.Text = "Your name is " + Request.Cookies["name"].Value;
    }
}

// нажатие на кнопку устанавливает cookie
protected void btn_Click(object sender, EventArgs e)
{
    // устанавливаем ненулевое время жизни (3 дня),
    // чтобы информация сохранилась после закрытия браузера
    HttpCookie ac = new HttpCookie("name")
    { Expires = DateTime.Now.AddDays(3), Value = nameTextBox.Text };
    Response.Cookies.Add(ac);
}
</script>
```

```
<html>
<body>
  <form id="form1" runat="server">
    <asp:Label ID="lbl" runat="server" /> <br />
    <asp:TextBox ID="nameTextBox" runat="server" /> <br />
    <asp:Button ID="btn" runat="server" Text="Set cookie"
                OnClick="btn_Click" />
  </form>
</body>
</html>
```

Кроме состояний сеансов и cookie, ASP.NET предоставляет средства хранения состояний отдельных клиентов с помощью механизма, называемого *состоянием представления*. Состояние представления храниться в скрытом поле `__VIEWSTATE` каждой страницы ASP.NET. При каждом запросе страницы самой себя содержимое поля `__VIEWSTATE` передается в составе запроса. Состояния представления предназначены главным образом для сохранения состояний элементов управления в последовательности ответных запросов, однако их можно использовать и как механизм сохранения состояний отдельных клиентов между ответными запросами к этой же странице.

Состояние представления доступно из любого элемента управления как свойство `ViewState`. Оно представляется как объект типа `StateBag`, поддерживающий хранение любых сериализуемых типов.

5.23. КЭШИРОВАНИЕ В ASP.NET

В ASP.NET различают два вида кэширования: *кэширование вывода* и *кэширование данных*. При задействованном кэшировании вывода страница генерируется только при первом обращении. При любом следующем обращении она будет возвращена из кэша, что экономит время на её генерацию. Кэширование данных позволяет запоминать в кэше произвольные объекты, причем для этих объектов можно установить время нахождения в кэше и зависимости, при нарушении которых объект из кэша удаляется.

Кэширование вывода управляется директивой `@OutputCache`. Атрибуты данной директивы и их описание представлены в табл. 41.

Таблица 41

Атрибуты директивы `@OutputCache`

Атрибут	Значение	Описание
`Duration`	Число	Время в секундах существования страницы или пользовательского элемента управления в кэше

`Location`	`Any`
	`Client`
	`Downstream`
	`Server`
	`ServerAndClient`
	`None`
		Атрибут управляет передаваемыми клиенту заголовком и метадескрипторами, указывающими, где может быть кэширована страница. При значении `Client` страница кэшируется только в браузере, при значении `Server` – только на веб-сервере, при значении `Downstream` – на прокси-сервере и у клиента. При `ServerAndClient` кэш располагается на

		клиенте или на веб-сервере, прокси-сервера для кэша не используются. None отключает кэширование страницы
VaryByCustom	browser Спец. строка	Страница кэшируется по-разному в зависимости от имени и версии браузера клиента или от заданной строки, обрабатываемой переопределенным методом GetVaryByCustomString()
VaryByHeader	* Имена заголовков	Список строк, представляющих передаваемые клиентам заголовки
VaryByParam	none * Имена параметров	Список строк, разделенных точкой с запятой и представляющих значения строки запроса GET или переменные запроса POST
VaryByControl	Список полностью квалифицированных имен свойств	Список строк, представляющих свойства пользовательского элемента управления, используемые для настройки кэшей вывода (атрибут применяется только с пользовательскими элементами управления)

В директиве @OutputCache атрибуты Duration и VaryByParam являются обязательными. Если VaryByParam="none", то для каждого типа запроса (GET, HEAD или POST) будет кэшироваться по одной копии страницы.

В листинге показан пример использования директивы @OutputCache, задающей существование страницы в кэше на протяжении одного часа после первого обращения. В странице выводится дата её генерации, поэтому во всех выводах после первого вы увидите одно и то же время, пока не истечёт срок жизни страницы.

```
<%@ Page Language="C#" %>
<%@ OutputCache Duration="3600" VaryByParam="none"%>
<script runat="server">
    void Page_Load(object sender, EventArgs e)
    {
        msg.Text = DateTime.Now.ToString();
    }
</script>

<html>
<body>
    <h3>Вывод кэшированной страницы</h3>
    <p>В последний раз страница сгенерирована:
    <asp:Label ID="msg" runat="server" /></p>
</body>
</html>
```

Запрос страницы может выполняться несколькими способами: при помощи глагола GET без параметров, при помощи глагола HEAD, при помощи GET с параметрами, или с использованием POST с телом, содержащим пары имя/значение. Кэшировать страницу, к которой обращаются с разными строками GET или переменными POST сложно, так как требуется кэшировать разные версии страницы для различных комбинаций параметров. Тонкую настройку кэширования в

этом случае можно провести при помощи атрибута VaryByParam директивы @OutputCache. Если задать VaryByParam="none", то кэш сохраняет одну версию страницы для каждого типа запроса. Получив запрос GET со строкой переменных, кэш вывода игнорирует строку и возвращает экземпляр страницы, кэшированный для GET. Если указать VaryByParam="*", будет кэширована отдельная версия страницы для каждой уникальной строки GET и каждого уникального набора параметров POST. Такой способ кэширования может оказаться крайне неэффективным. В атрибуте VaryByParam можно указать имя (или список имен) переменных запроса. В этом случае уникальные копии страницы будут сохраняться в кэше вывода, если в запросе значения перечисленных переменных разные.

Атрибуты VaryByHeader и VaryByCustom управляют сохранением версий страницы в кэше на основе значений заголовков HTTP-запроса и версий браузера соответственно. Например, если страница выводится по-разному в зависимости от заголовка Accept-Language, то нужно обеспечить кэширование отдельных копий для каждого языка, предпочитаемого клиентом:

```
<%@ OutputCache Duration="360" VaryByParam="none"
                VaryByHeader="Accept-Language" %>
```

В ASP.NET 2.0 появилась возможность установить зависимость кэшируемой страницы от таблицы в базе данных. Делается это с использованием нового атрибута директивы @OutputCache SqlDependency:

```
<%@ OutputCache Duration="15" VaryByParam="none"
                SqlDependency="Northwind:Employees" %>
```

Вывод страницы, содержащей эту директиву, будет кэшироваться в течение 15 секунд или до тех пор, пока в одну из записей таблицы Employees базы данных Northwind не будет внесено изменение. Заметьте, что Northwind здесь - не имя базы данных, а имя элемента в разделе <databases> конфигурационного файла. Этот элемент содержит ссылку на строку подключения к базе данных. Можно определить несколько зависимостей, разделив соответствующие пары *база:таблица* символами точки с запятой.

Рассмотрим вопросы, связанные с кэшированием данных. Для доступа к кэшу разработчик может использовать свойство Cache класса Page или объекта HttpContext. Область видимости кэша данных ограничена приложением. Его возможности во многом идентичны возможностям хранилища HttpApplicationState при двух важных отличиях. Во-первых, по умолчанию не гарантируется, что информация, размещенная в кэше данных, сохраняется в нем. Разработчик всегда должен быть готов к тому, что в кэше нет нужной информации. Во-вторых, кэш не предназначен для хранения обновляемых пользователями данных. Кэш позволяет читать данные многим потокам и записывать одному потоку, но не существует методов, подобных Lock() и UnLock(), позволяющих разработчику управлять блокировками. Следовательно, кэш данных предназначен для хранения данных в режиме «только для чтения» в целях облегчения доступа к ним.

Простейший вариант использования свойства Cache заключается в работе с ним как с коллекцией, аналогично работе со свойствами Session и Application. Однако при каждом добавлении некоторого объекта в кэш одновременно в кэш заносится экземпляр класса CacheEntry, описывающий кэшируемый объект. Основные свойства класса CacheEntry представлены в табл. 42. Обратите внимание на возможность связывания кэшируемых объектов с другими элементами кэша или файлами:

Таблица 42

Свойства класса CacheEntry

Свойство	Тип	Описание
Key	Строка	Уникальный ключ для идентификации раздела кэша
Dependency	CacheDependency	Зависимость данного раздела от файла, каталога или другого раздела (при их изменении раздел кэша будет удален)
Expires	DateTime	Фиксированные дата и время, при достижении которых раздел будет удален
SlidingExpiration	TimeSpan	Интервал времени между последним обращением к разделу и его удалением
Priority	CacheItemPriority	Важность сохранения данного раздела по сравнению с другими разделами (используется при чистке кэша)
OnRemoveCallback	CacheItem-RemovedCallback	Делегат, который может запускаться при удалении раздела

Если для помещения объекта в кэш используется индексатор по умолчанию, то свойства соответствующего объекта CacheEntry принимают следующие значения: время жизни устанавливается бесконечным, скользящее время жизни равно нулю, значение Priority равно Normal, а OnRemoveCallback – null. При этом объект в кэше может оставаться как угодно долго, пока его не удалит процедура очистки (обычно вследствие нехватки памяти) или пока он не будет удалён явно.

Если возникла необходимость управлять свойствами CacheEntry, то можно воспользоваться одной из перегруженных версий методов Insert() или Add(). В следующем примере показан файл global.asax, добавляющий содержимое файла в кэш при запуске приложения. Раздел кэша становится неправильным, когда изменяется содержимое файла, использованного для заполнения раздела, поэтому в кэш добавлен объект CacheDependency. В сценарии также зарегистрирован метод обратного вызова, позволяющий получить извещение об удалении данных из кэша. Кроме этого, для добавленного раздела отключен срок действия, отключен режим скользящего времени жизни и задан приоритет по умолчанию:

```
<%@ Application Language="C#" %>
<script runat="server">
public void OnRemovePi(string key, object val,
```

```
                        CacheItemRemovedReason r)
{
    // некоторые действия, выполняемые при удалении элемента
    // например, помещение элемента снова в кэш.
}

protected void Application_Start(object sender, EventArgs e)
{
    var fileName = Server.MapPath("pi.txt");
    var pi = new StreamReader(fileName).ReadToEnd();
    var piDep = new CacheDependency(fileName);
    Context.Cache.Add("pi", pi, piDep, Cache.NoAbsoluteExpiration,
                      Cache.NoSlidingExpiration,
                      CacheItemPriority.Default, OnRemovePi);
}
</script>
```

Любая страница этого приложения может обращаться к кэшу данных по ключу `pi`. Кроме этого, гарантируется согласованность раздела `pi` с содержимым файла `pi.txt`.

```
<!-- Файл: default.aspx -->
<%@ Page Language="C#" %>
<script runat="server">
protected void Page_Load(object src, EventArgs e)
{
    pi.Text = Cache["pi"] == null
              ? "Undefined!"
              : (string)Cache["pi"];
}
</script>

<html>
<body>
    <form runat="server">
        <h1> Страница pi </h1>
        <asp:TextBox ID="pi" runat="server" />
    </form>
</body>
</html>
```

Класс `CacheDependency` имеет несколько перегруженных конструкторов, позволяющих установить зависимости. Кроме этого, в ASP.NET 2.0 разработчик может описать наследник этого класса для установки нестандартных зависимостей. Новый класс ASP.NET 2.0 `SqlCacheDependency` наследует `CacheDependency` и поддерживает зависимости от таблиц MS SQL Server. Этот класс совместим с SQL Server 7.0 и последующими версиями SQL Server.

5.24. ОБЕСПЕЧЕНИЕ БЕЗОПАСНОСТИ В ВЕБ-ПРИЛОЖЕНИЯХ

Рассмотрим основные понятия, связанные с процессом обеспечения безопасности. *Аутентификацией* (*authentication*) называется процесс идентификации пользователей приложений. *Авторизация* (*authorization*) – это процесс предоставления доступа пользователям на основе их идентификационных данных. Аутентификация наряду с авторизацией представляют собой средства защиты веб-приложений от несанкционированного доступа.

ASP.NET совместно с веб-сервером обеспечивает несколько возможных типов аутентификации: `Windows` (по умолчанию), `Forms`, `Passport` и `None`[1]. Выбор типа определяет механизм хранения маркеров аутентифицированного пользователя. В случае типа `Windows` маркер помещается в контекст потока рабочего процесса ASP.NET. Если используется тип `Forms`, маркер передается от клиента к серверу и обратно в cookie-файле.

Задать требования аутентификации клиентов веб-приложения можно путем добавления соответствующих элементов в конфигурационный файл приложения `web.config`. Тип аутентификации клиентов задается с помощью элемента `<authentication>`, атрибут `mode` которого может принимать одно из четырех значений: `Windows`, `Forms`, `Passport` и `None`. Сконфигурировав тип аутентификации, нужно продумать цели аутентификации. Аутентификация выполняется для ограничения доступа клиентов ко всему приложению или к его частям с помощью элемента `<authorization>`. В этот элемент добавляются подэлементы `<allow>` и `<deny>`, задающие предоставление или отказ в доступе отдельным пользователям и ролям. Метасимвол `*` используется для представления всех пользователей, а `?` – для представления анонимных пользователей. Следующий пример конфигурационного файла отказывает анонимным пользователям в праве доступа к сайту:

```
<configuration>
  <system.web>
    <authorization>
      <deny users="?" />
    </authorization>
  </system.web>
</configuration>
```

Элементы `<allow>` и `<deny>` поддерживают три атрибута: `users`, `roles` и `verbs`. Значениями этих атрибутов могут быть разделенные запятыми списки пользователей, ролей и команд. Существует возможность определить отдельные параметры безопасности для выбранного подкаталога веб-приложения. Для этого в подкаталог помещается файл `web.config` с надлежащей секцией

[1] Методы аутентификации ASP.NET применяются для файлов, являющихся частью Web-приложения. HTML-страницы (`*.htm` или `*.html`) не включаются в число этих файлов автоматически. Такие страницы обрабатываются IIS, а не ASP.NET. Чтобы зарегистрировать HTML-страницы для обработки рабочим процессом ASP.NET, требуется соответствующим образом настроить IIS.

<authorization> или используются возможности конфигурационного раздела <location>. При определении прав доступа пользователя к ресурсу ASP.NET придерживается следующего алгоритма:

- Строится список всех <allow> и <deny> элементов, начиная с ближайшего к ресурсу web.config, а затем добавляя элементы из web.config более высокого уровня, и, наконец, элементы machine.config.
- Список просматривается сверху вниз до первого совпадения имени или роли в списке с именем или ролью текущего пользователя.
- Тот элемент, который будет обнаружен при совпадении, определяет, может ли пользователь получить доступ к ресурсу. То есть, если обнаружили при совпадении элемент <allow>, то может, если <deny> - то нет.

Отметим, что если клиент аутентифицируется, то информация о нем доступна посредством свойства User класса Page или класса HttpContext. Свойство User указывает на реализацию интерфейса IPrincipal. Этот интерфейс содержит одно свойство и один метод. Свойство Identity является указателем на реализацию интерфейса IIdentity, а метод IsInRole() проверяет членство клиента в указанной группе. Ниже приведено описание интерфейса IIdentity:

```
public interface IIdentity
{
    string AuthenticationType { get; }
    bool IsAuthenticated { get; }
    string Name { get; }
}
```

Свойство IsAuthenticated используется для различения аутентифицируемых и анонимных клиентов. Свойство Name можно применить для выяснения идентичности клиента. Если клиент аутентифицируется, то свойство AuthenticationType позволит выяснить тип аутентификации.

Вернемся к рассмотрению различных типов аутентификации и остановимся на режиме аутентификации Forms. Принцип действия аутентификации в этом режиме следующий:

1. Когда пользователь первый раз запрашивает ресурс, требующий аутентификации, сервер перенаправляет запрос в выделенную страницу регистрации.
2. Регистрационная страница принимает данные пользователя (как правило, имя и пароль), а затем приложение аутентифицирует пользователя (предположительно с помощью базы данных).
3. Если пользователь успешно зарегистрировался, сервер предоставляет ему аутентификационный файл cookie в зашифрованном виде, который действителен на протяжении сеанса (но может быть сохранен на клиентском компьютере для использования в последующих сеансах).

4. Пользователь перенаправляется к интересующему его ресурсу, однако теперь он предоставляет в запросе идентификационный cookie и получает доступ.

Если в файле `web.config` задать аутентификацию `Forms`, то дополнительные настройки режима можно выполнить во вложенном элементе `<forms>`, атрибуты которого содержит табл. 43.

Таблица 43

Атрибуты элемента `<forms>`

Атрибут	Значения	Значение по умолчанию	Описание
`name`	Строка	`.ASPXAUTH`	Имя файла cookie
`loginUrl`	Адрес URL	`login.aspx`	Адрес URL страницы регистрации
`protection`	`All`, `None`, `Encryption`, `Validation`	`All`	Режим защиты файла cookie
`timeout`	Минуты	`30`	Скользящее время жизни файла cookie (сбрасывается при каждом запросе)
`path`	Маршрут	`/`	Маршрут файла cookie

Пример файла `web.config`, конфигурирующего аутентификацию в режиме `Forms`, показан в следующем листинге:

```
<configuration>
  <system.web>
    <authorization>
      <deny users="?"/>
    </authorization>
    <authentication mode="Forms">
      <forms loginUrl="login.aspx"/>
    </authentication>
  </system.web>
</configuration>
```

Теперь, чтобы аутентификация на основе cookie заработала, нужно реализовать страницу регистрации. Для реализации страницы регистрации платформа ASP.NET предоставляет класс `FormsAuthentication`. Этот класс состоит главным образом из статических методов, управляющих аутентификационными файлами cookie. Например, чтобы предоставить такой файл клиенту, нужно вызвать метод `SetAuthCookie()`. Метод `RedirectFromLoginPage()` предоставляет cookie клиенту и перенаправляет клиента в запрошенную им страницу.

Приведем пример страницы регистрации. Обратите внимание на наличие флажка, позволяющего запомнить данные клиента для следующих сеансов.

```
<!-- Файл login.aspx -->
<%@ Page Language="C#" %>

<script runat="server">
    protected void OnClick_Login(object src, EventArgs e)
```

```
    {
        if ((m_usrname.Text == "Alex") && (m_pass.Text == "pass"))
        {
            FormsAuthentication.RedirectFromLoginPage(
                                m_usrname.Text, m_save_pass.Checked);
        }
        else
        {
            msg.Text = "Неверно. Попытайтесь еще раз";
        }
    }
</script>

<html>
<body>
    <form id="Form1" runat="server">
    <h2>Страница регистрации</h2>
    Имя пользователя:
    <asp:TextBox ID="m_usrname" runat="server" /><br />
    Пароль:
    <asp:TextBox ID="m_pass" TextMode="Password" runat="server" />
    <br />
    Запомнить пароль?
    <asp:CheckBox ID="m_save_pass" runat="server" /><br />
    <asp:Button ID="Button1" Text="Регистрация"
                OnClick="OnClick_Login" runat="server" /><br />
    <asp:Label ID="msg" runat="server" />
    </form>
</body>
</html>
```

Рассмотрим еще один пример. Достаточно часто клиенту предоставляется возможность идентифицировать себя, если он этого хочет, или возможность анонимного доступа в противном случае. В этом случае имеет смысл интегрировать форму регистрации в страницу. При регистрации вызывается метод `FormsAuthentication.SetAuthCookie()` и выполняется перенаправление на эту же страницу. При загрузке страницы проверяется, идентифицирован ли пользователь, и если да, то выводиться специфичная для пользователя информация. Текст страницы `default.aspx` представлен ниже:

```
<!-- Файл default.aspx -->
<%@ Page Language="C#" %>

<script runat="server">
    protected void Page_Load(object src, EventArgs e)
    {
        msg.Text = User.Identity.Name == "Alex"
                   ? "Hello Alex" : "General Info";
    }
```

```
        protected void OnClick_Login(object src, EventArgs e)
        {
            if ((m_usrname.Text == "Alex") && (m_pass.Text == "pass"))
            {
               FormsAuthentication.SetAuthCookie(m_usrname.Text, false);
            }
            Response.Redirect("default.aspx");
        }
    </script>

    <html>
    <body>
        <form id="Form1" runat="server">
        <h2>Регистрация</h2>
        Имя :
        <asp:TextBox ID="m_usrname" runat="server" /><br />
        Пароль:
        <asp:TextBox ID="m_pass" TextMode="Password" runat="server" />
        <br />
        <asp:Button ID="Button1" Text="Войти" OnClick="OnClick_Login"
                    runat="server" /><br />
        <asp:Label ID="msg" runat="server" />
        </form>
    </body>
    </html>
```

5.25. УПРАВЛЕНИЕ ЧЛЕНСТВОМ И РОЛЯМИ

Наиболее заметным изменением, внесенным в механизм аутентификации Forms в ASP.NET 2.0, является введение дополнительного API, а именно *API управления членством и ролями*. Он представляют собой набор классов, предназначенных для управления учетными записями пользователей и ролями.

Используемые совместно с классом `FormsAuthentication` новые классы `Membership` и `Roles` составляют полный арсенал необходимых разработчикам ASP.NET средств защиты. Класс `Membership` предоставляет методы для управления учетными записями пользователей, в частности для добавления учетных записей новых пользователей, а также для удаления и редактирования существующих записей. Класс `Roles` служит связующим звеном между пользователями и их ролями. По умолчанию классы `Membership` и `Roles` работают с поставщиком, который сохраняет информацию о пользователях в базе данных MS SQL Express в стандартном формате. Если вы желаете работать с пользовательским хранилищем данных, вам достаточно будет создать собственный поставщик и подключить его к приложению.

Перечень свойств класса `Membership`, сопровождаемый кратким описанием, приведен в табл. 44. Все свойства класса `Membership` являются статическими и доступными только для чтения.

Таблица 44

Свойства класса `Membership`

Свойство	Описание
`ApplicationName`	Строка, идентифицирующую приложение. По умолчанию содержит путь к его корневой папке
`EnablePasswordReset`	Возвращает значение `true`, если поставщик поддерживает сброс паролей
`EnablePasswordRetrieval`	Возвращает значение `true`, если поставщик поддерживает восстановление паролей
`MaxInvalidPasswordAttempts`	Максимальное количество попыток ввода пароля перед блокированием пользователя
`MinRequired-NonAlphanumericCharacters`	Минимальное число знаков препинания в пароле
`MinRequiredPasswordLength`	Минимальная длина пароля
`PasswordAttemptWindow`	Время в минутах, в течение которого пользователь может пытаться ввести пароль, прежде чем будет заблокирован
`PasswordStrength-RegularExpression`	Регулярное выражение, которому должен отвечать пароль
`Provider`	Экземпляр используемого поставщика
`Providers`	Коллекция зарегистрированных поставщиков
`RequiresQuestionAndAnswer`	Возвращает значение `true`, если при восстановлении или сбросе пароля поставщик требует ответа на определенный вопрос
`UserIsOnlineTimeWindow`	Возвращает время в минутах, истекшее после последнего действия пользователя, в течение которого пользователь все еще будет считаться подключенным

В табл. 45 перечислены методы, поддерживаемые классом `Membership`. Из этого перечня станет ясно, каковы его задачи.

Таблица 45

Методы класса `Membership`

Имя метода	Описание
`CreateUser()`	Создает новую учетную запись пользователя (или ничего не делает, если учетная запись заданного пользователя уже существует). Этот метод возвращает объект `MembershipUser`, содержащий всю доступную информацию о пользователе
`DeleteUser()`	Удаляет учетную запись заданного пользователя
`FindUsersByEmail()`	Возвращает коллекцию объектов `MembershipUser`, для которых задан указанный вами адрес электронной почты
`FindUsersByName()`	Возвращает коллекцию объектов `MembershipUser`, для которых задано указанное вами имя пользователя
`GeneratePassword()`	Генерирует случайный пароль заданной длины
`GetAllUsers()`	Возвращает коллекцию всех пользователей
`GetNumberOfUsersOnline()`	Возвращает количество подключенных в данный момент пользователей
`GetUser()`	Извлекает объект `MembershipUser`, связанный с текущим или заданным пользователем
`GetUserNameByEmail()`	Возвращает имя пользователя, имеющего заданный адрес

	электронной почты. Если несколько пользователей имеют один и тот же адрес, возвращается первый из них
`UpdateUser()`	Принимает объект `MembershipUser` и обновляет хранящуюся в приложении информацию о пользователе
`ValidateUser()`	Аутентифицирует пользователя по учетным данным

Перед тем, как продемонстрировать некоторые примеры кода, использующие класс `Membership`, рассмотрим вопросы, связанные с конфигурированием системы управлением членством. Будем предполагать, что применяется встроенный поставщик членства `SqlMembershipProvider`. В используемой приложением базе данных необходимо создать таблицы для поставщика. Затем требуется отредактировать файл `web.config` приложения. (Предполагается, что в нем уже имеются записи для метода аутентификации Web Forms). Следующую секцию необходимо поместить в раздел `<system.web>`.

```
<membership defaultProvider="SqlProvider">
  <providers>
    <clear />
    <add
      name="SqlProvider"
      type="System.Web.Security.SqlMembershipProvider"
      connectionStringName="MySqlConnection"
      applicationName="MyApplication"
      enablePasswordRetrieval="false"
      enablePasswordReset="true"
      requiresQuestionAndAnswer="true"
      requiresUniqueEmail="true"
      passwordFormat="Hashed" />
  </providers>
</membership>
```

Дадим некоторые комментарии. Атрибут `defaultProvider` указывает на имя поставщика членства. Атрибуты конфигурации поставщика описаны в табл. 46.

Таблица 46

Атрибуты конфигурации `SqlMembershipProvider`

Имя атрибута	По умолчанию	Описание
`connectionStringName`		Имя строки подключения из секции конфигурационного файла
`enablePasswordReset`	`False`	Указывает, возможно ли изменение паролей пользователем
`requiresQuestionAndAnswer`	`False`	Определяет, необходим ли секретный вопрос и ответ при изменении пароля
`applicationName`	`/`	Имя приложения. Позволяет группировать пользователей по этому признаку в БД. А значит, использовать одну базу для хранения аутентификационной информации нескольких приложений

requiresUniqueEmail	False	Определяет, должен ли указываемый пользователем e-mail быть уникальным в рамках одного веб-приложения
maxInvalidPasswordAttempts	5	Число неправильных попыток ввода пароля, после которых учетная запись пользователя блокируется
passwordAttemptWindow	10	Время в минутах, в течение которого пользователь может пытаться ввести пароль, прежде чем будет заблокирован
passwordFormat		Формат хранения пароля. Допустимы значения Clear, Encrypted и Hashed
passwordAnswerAttempt LockoutDuration	30	Время в минутах, в течение которого действует блокировка аккаунта при вводе неправильного пароля
minRequiredPasswordLength	7	Минимальная длина пароля (от 1 до 128)
minRequiredNon AlphanumericCharacters	1	Минимальное количество символов в пароле, которые не являются буквой или цифрой
passwordStrength RegularExpression	""	Регулярное выражение, которому должен отвечать пароль

Рассмотрим примеры кода. Начнем с наиболее простой операции - аутентификации. Применяя аутентификационные функции API членства, можно написать следующую функцию входа:

```
private void LogonUser(object sender, EventArgs e)
{
    string user = userName.Text;
    string pswd = password.Text;
    if (Membership.ValidateUser(user, pswd))
    {
        FormsAuthentication.RedirectFromLoginPage(user, false);
    }
    else
    {
        errorMsg.Text = "Sorry, not a valid account.";
    }
}
```

Для программного создания новой учетной записи достаточно написать `Membership.CreateUser(userName, pswd)`. Удаление учетной записи пользователя выполняется с помощью метода `Membership.DeleteUser()`.

Для получения информации об определенном пользователе используется метод `GetUser()`. Он принимает имя пользователя и возвращает объект `MembershipUser`.

```
MembershipUser user = Membership.GetUser("TheUser");
```

Получив объект `MembershipUser`, вы обладаете всей необходимой информацией о пользователе и можете программно изменять его пароль и прочие

сведения. Приложение обычно выполняет несколько операций над паролями: изменяет их, отправляет пользователям, сбрасывает (обычно с использованием протокола запрос-ответ). Например, следующий код изменяет пароль:

```
MembershipUser user = Membership.GetUser("TheUser");
user.ChangePassword(user.GetPassword(), newPswd);
```

Методу `ChangePassword()` передается старый пароль. В некоторых случаях можно предоставить пользователю возможность просто сбросить пароль (то есть удалить старый и автоматически сгенерировать новый), вместо того чтобы его изменять. Для этого используется метод `ResetPassword()`:

```
MembershipUser user = Membership.GetUser("TheUser");
string newPswd = user.ResetPassword();
```

Рассмотрим вопросы, связанные с API управления ролями. Для включения поддержки ролей необходимо добавить в файл `web.config` приложения следующую строку:

```
<roleManager enabled="true" />
```

С помощью ролей устанавливаются правила доступа к страницам и папкам. Например, следующий блок `<authorization>` определяет, что только члены роли `Admin` имеют доступ к страницам, на которые распространяется действие файла `web.config`:

```
<authorization>
  <allow roles="Admin" />
  <deny users="*" />
</authorization>
```

В ASP.NET применяется API управления ролями, представленный членами класса `Roles`. Когда управление ролями включено, ASP.NET создает экземпляр этого класса и добавляет его в контекст каждого запроса, то есть в объект `HttpContext`. Следующий код показывает, как программным способом создать роли `Admin` и `Guest` и заполнить их именами пользователей:

```
Roles.CreateRole("Admin");
Roles.AddUserToRole("TheUser", "Admin");
Roles.CreateRole("Guest");
var guests = new[] { "UserX", "Godzilla" };
Roles.AddUsersToRole(guests, "Guest");
```

Во время выполнения страницы информация о запросившем ее пользователе и его ролях доступна через объект `User` контекста HTTP. Следующий код показывает, как установить, принадлежит ли пользователь к определенной роли, и включить соответствующие функции:

```
if (User.IsInRole("Admin"))
// включаем функции, специфические для данной роли
```

Класс `Roles` обладает удобным интерфейсом, позволяющим работать как с отдельными пользователями и ролями, так и с их группами. Методы этого класса описаны в табл. 47.

Таблица 47

Методы класса `Roles`

Имя метода	Описание
`AddUsersToRole()`	Включает группу пользователей в состав роли
`AddUsersToRoles()`	Включает группу пользователей в группу ролей
`AddUserToRole()`	Включает пользователя в состав роли
`AddUserToRoles()`	Включает пользователя в состав группы ролей
`CreateRole()`	Создает новую роль
`DeleteCookie()`	Удаляет cookie, которые менеджер ролей использовал для кэширования всех данных указанной роли
`DeleteRole()`	Удаляет роль
`FindUsersInRole()`	Извлекает имена пользователей, принадлежащих к заданной роли
`GetAllRoles()`	Возвращает список всех доступных ролей
`GetRolesForUser()`	Возвращает строковый массив ролей, к которым принадлежит заданный пользователь
`GetUsersInRole()`	Возвращает строковый массив с именами пользователей, принадлежащих к заданной роли
`IsUserInRole()`	Определяет, принадлежит ли указанный пользователь заданной роли
`RemoveUserFromRole()`	Удаляет пользователя из заданной роли
`RemoveUserFromRoles()`	Удаляет пользователя из группы заданных ролей
`RemoveUsersFromRole()`	Удаляет пользователей из заданной роли
`RemoveUsersFromRoles()`	Удаляет пользователей из группы заданных ролей
`RoleExists()`	Возвращает значение `true`, если заданная роль существует

Свойства класса `Roles` перечислены в табл. 48. Все они являются статическими и доступны только для чтения. Значения этих свойств соответствуют установкам из конфигурационного раздела `<roleManager>`.

Таблица 48

Свойства класса `Roles`

Свойства	Описание
`ApplicationName`	Возвращает имя приложения
`CacheRolesInCookie`	Возвращает значение `true`, если хранение данных ролей в cookie разрешено
`CookieName`	Определяет имя cookie, используемого для хранения информации о ролях
`CookiePath`	Определяет путь, ассоциированный с ролевым cookie
`CookieProtectionValue`	Содержит установку шифрования ролевого cookie: `All`, `Clear`, `Hashed` или `Encrypted`
`CookieRequreSSL`	Указывает, должны ли ролевые cookie передаваться через SSL-соединение
`CookieSlidingExpiration`	Указывает, является срок действия ролевого cookie фиксированным или скользящим

`CookieTimeout`	Возвращает срок действия ролевого cookie в минутах
`CreatePersistentCookie`	Создает ролевой cookie, сохраняющийся в течение сеанса
`Domain`	Определяет домен ролевого cookie
`Enabled`	Указывает, включена ли функция управления ролями
`MaxCachedResults`	Определяет максимальное количество ролей, которое может быть сохранено для одного пользователя в cookie-файле
`Provider`	Возвращает текущий поставщик ролей
`Providers`	Возвращает список поддерживаемых поставщиков ролей

Некоторым методам класса `Roles` необходима информация о ролях пользователя, поэтому ее обычно кэшируют в зашифрованных cookie. Получив очередной HTTP-запрос, ASP.NET проверяет, имеется ли в его составе ролевой cookie, и если имеется, дешифрует билет роли и присоединяет ее информацию к объекту `User`. По умолчанию ролевой cookie является сеансовым и становится недействительным, как только пользователь закрывает браузер.

Заметьте, что cookie содержит сведения о ролях лишь того пользователя, от которого поступил запрос. Когда вы запрашиваете информацию о ролях других пользователей, она считывается поставщиком ролей из источника данных.

Для выполнения ввода-вывода данных о ролях пользователей менеджер ролей использует модель поставщиков. Пользовательские поставщики ролей создаются как производные от класса `RoleProvider` и регистрируются в подразделе `<providers>` раздела `<roleManager>`. Заметьте, что этот процесс практически идентичен процессу создания пользовательских поставщиков членства.

В дополнение к API членства и ролей ASP.NET предоставляет в распоряжение разработчиков несколько серверных элементов управления, которые значительно упрощают разработку составляющих веб-приложения, обеспечивающих его защиту. Это элементы `Login`, `LoginName`, `LoginStatus`, `LoginView`, `PasswordRecovery`, `ChangePassword` и `CreateUserWizard`. Эти элементы инкапсулируют типичные код и разметку, которые до сих пор разработчику приходилось повторять в каждом приложении.

Составной элемент управления `Login` содержит полный комплекс элементов, типичных для формы входа. Этот элемент поддерживает также дополнительные элементы для восстановления забытого пароля, регистрации нового пользователя, вывода сообщений об ошибках и выполнения пользовательского действия при успешном входе. Перетащив элемент управления `Login` в форму в VS 2008, можно выбрать для него один из нескольких предопределенных стилей, или конвертировать его представление в шаблон, и затем описать части шаблона самостоятельно.

Маленький, но весьма полезный элемент управления `LoginName` - это обыкновенная надпись, в которой выводится имя пользователя. Он считывает имя пользователя из внутреннего объекта ASP.NET `User` и выводит с применением заданного стиля. Программный интерфейс элемента управления `LoginName` состоит из единственного свойства, `FormatString`, определяющего формат выводимого текста.

В элементе управления LoginStatus отображается состояние аутентификации текущего пользователя. Его пользовательский интерфейс состоит из кнопки ссылочного типа, предназначенной для входа в приложение либо выхода из него, смотря в каком состоянии находится пользователь.

Функцией элемента управления LoginView является определение блоков пользовательского интерфейса, выводимых для всех пользователей, которые принадлежат к определенной роли. Рассмотрим пример разметки с использованием LoginView, которая получена в VS 2008 при использовании команд контекстного меню элемента.

```
<asp:LoginView ID="LoginView1" runat="server">
  <RoleGroups>
    <asp:RoleGroup Roles="Admin,Manager">
      <ContentTemplate>
        <asp:Button ID="btn1" runat="server" Text="Button" />
      </ContentTemplate>
    </asp:RoleGroup>
    <asp:RoleGroup Roles="User">
      <ContentTemplate>
        <asp:Label ID="lbl1" runat="server" Text="Label"/>
      </ContentTemplate>
    </asp:RoleGroup>
  </RoleGroups>
  <AnonymousTemplate>
    <asp:Label ID="lbl2" runat="server" Text="Some text"/>
  </AnonymousTemplate>
</asp:LoginView>
```

Смысл разметки: для пользователей, которые не аутентифицированы, выводится содержимое блока <AnonymousTemplate>, для пользователей в ролях Admin и Manager показывается кнопка, а для пользователей в роли User – текст.

PasswordRecovery — это еще один серверный элемент управления, инкапсулирующий готовый фрагмент пользовательского интерфейса, который вы, в принципе, могли бы составить и сами. Он представляет форму, служащую для восстановления или сброса пароля. Пользователь получает новый или восстановленный пароль по электронной почте, в сообщении, направленном на указанный им при регистрации адрес. Элемент управления PasswordRecovery поддерживает три представления, выводимых на разных стадиях процесса восстановления пароля. Первое представление отображается, когда пользователь должен ввести свое имя, после чего элемент управления запрашивает у поставщика членства соответствующий объект MembershipUser. Второе представление предлагает пользователю ответить на контрольный вопрос, без чего не может быть получен пароль или осуществлен его сброс. Наконец, третье представление информирует пользователя о результатах операции. Элемент управления PasswordRecovery имеет дочерний элемент MailDefinition. В нем конфигурируется сообщение электронной почты, в частности указываются отправитель, формат тела сообщения (текст или HTML), приоритет, тема и получатель ко-

пии. Эти установки можно задать и программно с помощью соответствующих свойств класса.

Элемент управления `CreateUserWizard` предоставляет пользователю возможность создать и сконфигурировать новую учетную запись. Вы можете дать элементу управления указание отправлять новому пользователю уведомление по электронной почте. Адаптация элемента управления к нуждам конкретного приложения осуществляется в двух направлениях: настройки стандартных шагов мастера и добавления новых, служащих для ввода дополнительных сведений, таких как адрес, номер телефона, роли и т. п.

5.26. ПРОФИЛИ ПОЛЬЗОВАТЕЛЯ

Подсистема поддержки пользовательских профилей позволяет выполнять *персонализацию страниц*, то есть настройку страницы под конкретного пользователя. Для программиста пользовательский *профиль* выступает как набор свойств, которые сгруппированы в динамически сгенерированный класс. Объект этого класса доступен через свойство `Profile` страницы, а для постоянного хранения данных класса используется поставщик профиля (один из стандартных или специально разработанный).

Рассмотрим работу с профилями в предположении, что используется стандартный поставщик `SqlProfileProvider`. Естественно, для работы поставщика должен быть создан необходимый набор таблиц в БД приложения. Затем необходимо зарегистрировать `SqlProfileProvider` в файле `web.config`. Для этого необходимо определить строку подключения к базе профилей. Потом следует использовать раздел `<profile>` для удалений существующих поставщиков (элементом `<clear>`) и добавить поставщик `SqlProfileProvider`. Вот как должны выглядеть необходимые конфигурационные установки:

```
<system.web>
    <profile defaultProvider="SqlProvider">
      <providers>
        <clear />
        <add name="SqlProvider"
             type="System.Web.Profile.SqlProfileProvider"
             connectionStringName="SqlServices"
             applicationName="TestApplication" />
      </providers>
    </profile>
</system.web>
```

Когда вы определяете поставщик, то должны указать имя, тип, строку подключения (ссылку на соответствующий элемент в конфигурационном файле) и имя веб-приложения.

После регистрации поставщика профилей необходимо определить состав свойств профиля. Это делается добавлением элемента `<properties>` внутрь раздела `<profile>`. В элементе `<properties>` вы размещаете по одному дескрипто-

ру <add> для каждого специфичного свойства профиля. Как минимум, элемент <add> поддерживает имя свойства:

```
<profile defaultProvider="SqlProvider">
  <providers>
    . . .
  </providers>
  <properties>
    <add name="FirstName"/>
    <add name="LastName"/>
  </properties>
</profile>
```

Обычно также указывается тип данных (если этого не сделать, предполагается, что тип свойства - строка). Можно специфицировать любой сериализуемый класс в качестве типа (`String`, `Int32`, `DateTime`). Можно также установить еще несколько атрибутов, чтобы создать более расширенные свойства. Эти атрибуты перечислены в табл. 49

Таблица 49

Атрибуты свойств профилей

Имя атрибута	Описание
`name`	Имя свойства
`type`	Полное квалифицированное имя класса, представляющего тип свойства (по умолчанию – `String`)
`serializeAs`	Формат, используемый при сериализации свойства (`String`, `Binary`, `Xml` или `ProviderSpecific`)
`readonly`	Если установлен в `true`, то свойство может быть прочитано, но не изменено (попытка изменить свойство вызовет ошибку времени компиляции). По умолчанию имеет значение `false`
`defaultValue`	Значение по умолчанию, которое будет использовано, если профиль не существует либо не включают определенного фрагмента информации. Значение по умолчанию не затрагивает сериализацию – свойства профиля записываются в БД, даже если их значения равны `defaultValue`
`allowAnonymous`	Булево значение, указывающее, должно ли свойство применяться со средствами анонимных профилей. По умолчанию равно `false`
`provider`	Поставщик профилей, который должен быть использован для управления только данным свойством. По умолчанию все свойства управляются с применением поставщика, указанного в элементе <profile>, но можно назначить для разных свойств разных поставщиков

Поскольку профиль привязан к пользователю, вы должны аутентифицировать текущего пользователя перед тем, как читать или писать информацию профилей. Необходимо добавить правило авторизации, предотвращающее анонимный доступ к странице или папке, с которыми вы планируете использовать профили.

Когда запускается приложение, ASP.NET создает новый класс для представления профиля, наследуя его от `System.Web.Profile.ProfileBase`, который служит оболочкой для коллекции установок профиля. ASP.NET добавляет по

одному строго типизированному свойству к этому классу для каждого свойства профиля, определенного в файле `web.config`. Например, если определить строковое свойство по имени `FirstName`, то его значение на странице можно устанавливать следующим образом:

```
Profile.FirstName = "Alex";
```

Технически, полный профиль извлекается из постоянного хранилища, когда код в первый раз обращается к какому-либо свойству профиля. Информация профиля сбрасывается в базу данных по окончании запроса страницы. Если вы хотите сохранить профиль раньше, просто вызовите метод `Profile.Save()`.

Если у вас есть большое количество установок в профиле и некоторые из них логически связаны между собой, то можно использовать группы свойств для достижения лучшей организации. Например:

```
<profile defaultProvider="SqlProvider">
  <properties>
    <group name="Preferences">
      <add name="FirstName"/>
      <add name="LastName"/>
    </group>
    <group name="Address">
      <add name="Street" type="String" />
      <add name="City" type="String" />
    </group>
  </properties>
</profile>
```

После этого вы можете обращаться к свойствам из своего кода, используя имя группы: `lblCity.Text = Profile.Address.City`. Группы - это всего лишь замена полноценного пользовательского класса или структуры. Того же эффекта можно достичь, объявив собственный класс `Address`. Условие для такого класса – он должен быть сериализуем.

Для выполнения задач с профилями других пользователей можно воспользоваться классом `ProfileManager` (пространство имен `System.Web.Profile`), который предоставляет удобные статические методы, перечисленные в табл. 50. Многие из этих методов работают с классом `ProfileInfo`, который предоставляет информацию о профиле: имя пользователя (`UserName`), даты последнего обновления и последней активности (`LastUpdateDate` и `LastActivityDate`), размер профиля в байтах (`Size`), а также признак принадлежности анонимному пользователю (`IsAnonymous`). `ProfileInfo` не предоставляет действительных значений самого профиля.

Таблица 50

Статические методы класса `ProfileManager`

Имя метода	Описание
`DeleteProfile()`	Удаляет профиль указанного пользователя
`DeleteProfiles()`	Принимает массив имен пользователей и удаляет их профили
`DeleteInactiveProfiles()`	Удаляет профили, которые не использовались с указанного времени. Необходимо также передать значение из перечисления `AuthenticationOption`, чтобы указать, профили какого типа необходимо удалить (`All`, `Anonymous` или `Authenticated`)
`GetNumberOfProfiles()`	Возвращает количество записей с профилями в хранилище данных
`GetNumberOfInactiveProfiles()`	Возвращает количество записей с профилями, которые не использовались с указанного времени
`GetAllInactiveProfiles()`	Возвращает информацию профилей, которые не использовались с указанного времени. Профили возвращаются как объекты `ProfileInfo`
`GetAllProfiles()`	Извлекает все данные профилей из источника данных как коллекцию объектов `ProfileInfo`. Можно выбрать тип извлекаемых профилей (`All`, `Anonymous` или `Authenticated`). Существует перегруженная версия метода, использующая разбиение информации на страницы
`FindProfilesByUserName()`	Извлекает коллекцию объектов `ProfileInfo`, которые соответствуют определенному имени пользователя. `SqlProfileProvider` использует конструкцию `LIKE` при попытке найти соответствующее имя пользователя. Это значит, что можно применять шаблоны наподобие символа `%`. Существует перегруженная версия с разбиением на страницы
`FindInactiveProfilesByUserName()`	Извлекает информацию о профилях, которые не использовались с указанного времени. Можно отфильтровать определенные типы профилей (`All`, `Anonymous` или `Authenticated`) или имена пользователей, сравнивая с шаблоном

5.27. ЛОКАЛИЗАЦИЯ И РЕСУРСЫ

Под *ресурсом* в данном параграфе понимается строка, изображение или данные иного типа, хранящиеся отдельно от исполняемых файлов (кода). Ресурсы позволяют изменять вид приложения, не производя его перекомпиляцию и, прежде всего, используются в задачах *локализации*, то есть адаптации вида приложения под конкретный язык или культуру. ASP.NET предлагает унифицированный подход для работы с ресурсами, в котором центральную роль играет соответствующий API и файлы ресурсов.

Файл ресурсов – это обычный XML-файл, для которого принято использовать расширение `*.resx`. В файле хранятся данные в виде набора пар «имя ре-

сурса-значение ресурса». Visual Studio позволяет редактировать файлы ресурсов, скрывая внутренние детали их реализации.

В .NET Framework имеется набор классов, позволяющих прочитать файл ресурсов и извлечь из него требуемые значения. При использовании ASP.NET работа с ресурсами упрощена. Прежде всего, заметим, что с точки зрения приложения ASP.NET ресурсы делятся на *глобальные* и *локальные*. Глобальные ресурсы приложения хранятся в специальном каталоге приложения `App_GlobalResources`. Пусть, например, в этот каталог помещен файл `Resource.resx`, содержащий строковый ресурс `name` со значением `Alex`. Тогда доступ к этому ресурсу из кода страницы или из программного кода может быть выполнен так:

```
tbxSearch.Text = Resources.Resource.name;
```

Можно считать что ASP.NET создает для каждого глобального ресурса строго типизированный класс, имя которого совпадает с именем файла ресурсов (в нашем случае – `Resource`), а сами ресурсы доступны как свойства этого класса. Сгенерированные классы находятся в пространстве имен `Resources`.

В именовании файлов ресурсов имеется один нюанс, связанный непосредственно с локализацией. Дело в том, что исполняющая среда пытается найти файл ресурсов, связанный с контекстом культуры потока выполнения. Предполагается, что само имя файла непосредственно содержит указание на идентификатор культуры. Так, в предыдущем примере, имя `Resource.resx` обозначает ресурсы, используемые по умолчанию (если соответствие не будет найдено), имя `Resource.en.resx` обозначает ресурсы англоязычных стран, а имя `Resource.en-US.resx` – ресурсы для английского языка США. Контекст культуры для элементов управления определяет свойство `CurrentUICulture` класса `Thread`. Таким образом, при наличии файла `Resource.de-DE.resx` с соответствующим свойством `name`, следующий код будет использовать немецкий ресурс:

```
Thread.CurrentThread.CurrentUICulture = new CultureInfo("de-DE");
tbxSearch.Text = Resources.Resource.name;
```

В реальных веб-приложениях информацию о культуре передает браузер клиента и она подобным образом обычно не устанавливается. Можно определить регион, используемый по умолчанию, в файле `web.config`:

```
<globalization enableClientBasedCulture="true"
               culture="de-DE" uiCulture="de-DE"/>
```

Для указания региона для отдельной страницы можно применить директиву страницы и её атрибуты `Culture` и `UICulture`.

В ASP.NET 2.0 работа с ресурсами страниц еще более облегчилась. При создании страницы в Visual Studio достаточно переключится на дизайнер страниц и выполнить команду `Tools | Generate Local Resources`. Для страницы

будет автоматически выполнена генерация файла локальных ресурсов. Он размещается в папке App_LocalResources, и его имя – такое же как у страницы, но с расширением .resx. Имена ресурсов в этом файле построены по схеме: «*имя элемента управления* Resource *номер.имя свойства*». А если обратиться к коду разметки страницы, то у элементов управления можно заметить специальные атрибуты meta:resourcekey:

```
<asp:ImageButton ID="btnSearchGo" runat="server"
                 meta:resourcekey="btnSearchGoResource1" />
```

Несложно понять, что именно при помощи этих атрибутов связывается элемент управления и группа ресурсов с соответствующим именем в файле ресурсов. Все последующие ресурсы, специфические для данного региона, придется добавлять вручную, копируя сгенерированные ресурсы и присваивая им соответствующее имя (например, Default.aspx.en-US.resx).

Если необходимо локализовать не элемент управления на странице, а статический текст страницы, то следует воспользоваться новым элементом управления Localize. Он должен охватывать статический текст, тогда эта часть страницы будет автоматически включена в процесс генерирования ресурсов.

5.28. ПОЛЬЗОВАТЕЛЬСКИЕ ЭЛЕМЕНТЫ УПРАВЛЕНИЯ

Среда ASP.NET позволяет программисту создавать собственные элементы управления. Выделяют два вида элементов: *пользовательские элементы управления* (*User Controls, UC*) и *серверные элементы* (*Server Custom Controls, SCC*). Фактически, UC представляют собой фрагменты обыкновенной aspx-страницы, помещенные в специальную именованную оболочку. Процесс их создания в целом подобен процессу визуального дизайна. SCC – это полноценные классы, размещенные в отдельных сборках. Процесс создания SCC является невизуальным, и возможностей для тонкой настройки они предоставляют больше.

Пользовательский элемент управления может быть создан декларативно в текстовом или HTML редакторе. Декларативный синтаксис пользовательского элемента управления очень похож на синтаксис, используемый при создании страниц. Основным отличием является то, что пользовательский элемент управления не включает элементы <html>, <body> и <form>. В качестве примера UC приведём компонент в виде текстовой метки и поля для ввода. Ниже помещён код компонента (файл Input.ascx[1]).

```
<%@ Control Language="C#" %>

<script language="C#" runat="server">
    public string LabelText
    {
        get { return lbl.Text; }
```

[1] В Visual Studio можно начать новый web-проект, добавить в проект новый элемент Web User Control (файл LogonForm.ascx) и размещать элементы в дизайнере или ввести код.

```
            set { lbl.Text = value; }
        }
        public string InputText
        {
            get { return tbx.Text; }
            set { tbx.Text = value; }
        }
    </script>

    <table style="border: black 1px solid; font: 10pt verdana"
                  cellspacing="15">
        <tr>
            <td>
                <asp:Label ID="lbl" runat="server" />
            </td>
            <td>
                <asp:TextBox ID="tbx" runat="server" Width="144px" />
            </td>
        </tr>
    </table>
```

Создадим страницу, на которой будет использоваться элемент управления. При написании такой страницы необходимо использовать директиву `@Register` для указания следующих параметров:

- `Src` – имя `ascx`-файла, содержащего элемент управления;
- `TagName` – имя тэга, идентифицирующего элемент управления;
- `TagPrefix` – префикс для тэгов, ссылающихся на элемент управления.

```
<%@ Register TagPrefix="abc" TagName="InputControl"
             Src="~\Input.ascx" %>
<html>
<body>
    <form id="Form1" method="post" runat="server">
    <abc:InputControl id="input1" runat="server" />
    </form>
</body>
</html>
```

Вообще говоря, существует универсальный способ преобразования любой `aspx`-страницы в пользовательский элемент управления. Чтобы сделать это:

1. Удалите все элементы `<html>`, `<body>` и `<form>` из страницы.
2. Если в `aspx`-странице есть директива `@Page`, замените ее директивой `@Control`. Чтобы избежать ошибки синтаксического разбора при преобразовании страницы в элемент управления, удалите все атрибуты, поддерживаемые директивой `@Page`, которые не поддерживаются директивой `@Control`.

3. Включите атрибут `className` в директиву `@Control` (при желании). Это даст возможность присвоить строгий тип пользовательскому элементу управления при программном добавлении на страницу или к другому серверному элементу управления.
4. Присвойте элементу управления имя файла, которое отражает планы по его использованию, и измените расширение имени файла на `.ascx`.

UC похож на страницу и тем, что он может содержать обработчики собственных событий. Следующий пример демонстрирует элемент управления, инкапсулирующий `LinkButton`, который отображает текущее время. При щелчке на ссылке или загрузке страницы время обновляется. При разработке этого элемента управления поместим требуемый код в файл Code Behind. Также определим собственное свойство, отвечающее за формат отображения времени.

```
<%@ Control CodeBehind="TimeDisplay.ascx.cs"
            Inherits="TimeDisplay" %>

<asp:LinkButton id="lnkTime" runat="server"
                OnClick="lnkTime_Click" />

using System;
using System.Web.UI;
using System.Web.UI.WebControls;

public class TimeDisplay : UserControl
{
    protected LinkButton lnkTime;
    private string format;

    public string Format
    {
        get { return format; }
        set { format = value; }
    }

    protected void Page_Load(object sender, EventArgs e)
    {
        if (!Page.IsPostBack)
        {
            RefreshTime();
        }
    }

    protected void lnkTime_Click(object sender, EventArgs e)
    {
        RefreshTime();
    }

    public void RefreshTime()
```

```
        {
            lnkTime.Text = string.IsNullOrEmpty(format)
                                ? DateTime.Now.ToLongTimeString()
                                : DateTime.Now.ToString(format);
        }
    }
```

Рассмотрим процесс создания серверного элемента управления. По сути, создание такого элемента представляет собой разработку пользовательского класса. Естественно, что такой класс не создается «с нуля», а является наследником некоторых стандартных классов:

- `System.Web.UI.Control` – базовый класс для любого (пользовательского или стандартного) элемента управления;
- `System.Web.UI.WebControls.WebControl` – класс, содержащий методы и свойства для работы со стилем представления. Этот класс является потомком класса `System.Web.UI.Control`;
- `System.Web.UI.HtmlControls.HtmlControl` – базовый класс для стандартных элементов HTML, таких как `input`;
- `System.Web.UI.TemplateControl` – базовый класс для страниц и пользовательских элементов управления User Controls.

Построим простейший серверный элемент управления. Создадим библиотеку классов и включим в нее следующий код:

```
using System.Web.UI;

namespace MyControls
{
    public class FirstControl : Control
    {
        protected override void Render(HtmlTextWriter writer)
        {
            writer.Write("<h1>ASP.NET Custom Control</h1>");
        }
    }
}
```

Наш элемент управления очень прост. Это класс `FirstControl`, в котором перекрыт виртуальный метод `Render()`. Именно этот метод вызывается исполняющей средой при необходимости отрисовки страницы и всех элементов управления, расположенных на ней. В методе `Render()` выполняется вывод HTML-кода при помощи средств объекта класса `System.Web.UI.HtmlTextWriter`.

При написании страницы, на которой будет использоваться элемент `FirstControl`, необходимо использовать директиву `@Register` для указания следующих параметров:

- `Assembly` – имя сборки со скомпилированным серверным элементом управления. Если указано слабое имя сборки, то она должна быть расположена в подкаталоге `Bin` приложения. Данный атрибут не надо использовать, если код элемента управления размещен в специальном каталоге `App_Code`;
- `Namespace` – имя пространства имен, содержащего написанный класс;
- `TagPrefix` – префикс для тэгов, ссылающихся на элемент управления.

Как вы уже знаете, настройка любого элемента управления при размещении его на странице выполняется при помощи атрибутов. ASP.NET преобразует значения атрибутов в значения соответствующих открытых свойств класса, представляющего элемент управления. Если тип свойства строковый, то происходит простое копирование значения атрибута в свойство. В случае числовых или булевских типов выполняется преобразование строки в соответствующий тип. Если тип свойства – некое перечисление, то строка атрибута задает имя элемента этого перечисления. Особый случай – использование в качестве типа свойств классов. Для доступа к свойству агрегированного объекта требуется применять формат «`Имя сложного свойства-Имя подсвойства`». Например:

```
<myControls:Control runat="server" Font-Color="Red"/>
```

Добавим некоторые свойства в класс `FirstControl`. Свойство `Text` будет содержать отображаемый текст, свойство `RepeatCount` – это количество повторений текста, свойство `ForeColor` – цвет текста (перечисление `System.Drawing.Color`). Заметим, что виртуальный метод `OnInit()` элемента управления вызывается после задания свойств, указанных в разметке страницы. Его можно переопределить для задания значения свойств по умолчанию или для контроля диапазона значений свойства.

```
using System;
using System.Drawing;
using System.Web.UI;

namespace MyControls
{
    public class FirstControl : Control
    {
        public string Text { get; set; }

        public int RepeatCount { get; set; }

        public Color ForeColor { get; set; }

        public FirstControl()
        {
            ForeColor = Color.Blue;
            RepeatCount = 1;
```

```
            Text = "Default Text";
        }

        protected override void OnInit(EventArgs e)
        {
            base.OnInit(e);
            if ((RepeatCount < 1) || (RepeatCount > 10))
            {
                throw
                    new ArgumentOutOfRangeException("RepeatCount");
            }
        }

        protected override void Render(HtmlTextWriter writer)
        {
            for (var i = 1; i <= RepeatCount; i++)
            {
                writer.Write("<h1 style='color:" +
                             ColorTranslator.ToHtml(ForeColor) +
                             "'>" + Text + "</h1>");
            }
        }
    }
}
```

При сложной схеме отображения, применяемой в методе `Render()`, удобным является использование дополнительных методов класса `HtmlTextWriter`, упрощающих построение HTML-кода. Эти методы перечислены в табл. 51.

Таблица 51

Методы класса `HtmlTextWriter`

Имя метода	Описание
`AddAttribute()`	Добавляет атрибут к следующему элементу HTML, который будет сформирован
`AddStyleAttribute()`	Добавляет соответствующий элемент для атрибута `style`
`RenderBeginTag()`	Формирует открывающий тег HTML-элемента
`RenderEndTag()`	Формирует замыкающий тег HTML-элемента и записывает этот элемент, а также любые атрибуты, формирование которых еще не закончено. После вызова этого метода все атрибуты считаются сформированными
`Write()`	Немедленно записывает строку
`WriteAttribute()`	Немедленно записывает HTML-атрибут
`WriteBeginTag()`	Немедленно записывает открывающий тег HTML-элемента
`WrileEndTag()`	Немедленно записывает замыкающий тег HTML-элемента
`WriteFullBeginTag()`	Немедленно записывает начальный тег вместе с замыкающей скобкой (>)
`WriteLine()`	Немедленно записывает строку содержимого. Эквивалент метода `Write()`, отличающийся лишь добавлением к концу строки символа перехода на новую строку

Применяя методы класса HtmlTextWriter, FirstControl.Render() можно переписать следующим образом:

```
protected override void Render(HtmlTextWriter writer)
{
    for (var i = 1; i <= RepeatCount; i++)
    {
        writer.AddStyleAttribute(HtmlTextWriterStyle.Color,
                                 ColorTranslator.ToHtml(ForeColor));
        writer.RenderBeginTag("h1");
        writer.Write(Text);
        writer.RenderEndTag();
    }
}
```

Если пользовательский элемент управления использует стили отображения, то имеет смысл рассмотреть в качестве базового класса для такого элемента класс WebControl из пространства имен System.Web.UI.WebControls. Класс WebControl не только включает базовые свойства стиля - такие как Font, ForeColor, BackColor и т. п., но он также отображает их автоматически в соответствующем теге HTML. Вот как это работает: WebControl предполагает, что он должен добавить атрибуты к одному тегу HTML, называемому *базовым*. Если вы пишете множественные элементы, атрибуты добавляются к самому внешнему тегу, который содержит все остальные элементы. Базовый тег вашего веб-элемента указывается в конструкторе.

В итоге вы не переопределяете метод Render(). WebControl уже включает реализацию Render(), которая поручает работу следующим трем методам:

- RenderBeginTag(). Этот метод вызывается для вывода открывающего тега вашего элемента управления, вместе с указанными атрибутами.
- RenderContents(). Этот метод выводит все, что находится между открывающим и закрывающим тегами. Это метод, который вы будете чаще всего переопределять.
- RenderEndTag(). Метод вызывается для вывода закрывающего тега HTML.

Если пользовательский элемент управления должен сохранять состояние, нужно использовать его внутреннюю коллекцию ViewState (или ControlState). Обычно работа с коллекцией происходит в методах свойств – аксессор читает данные, мутатор помещает их в коллекцию, снабдив ключом.

Если разрабатывается элемент управления, который позволяет вводить информацию, то такой элемент должен реализовывать интерфейс IPostBackDataHandler. Реализуя этот интерфейс, вы сообщаете ASP.NET о том, что когда происходит обратная отправка, ваш элемент управления должен иметь возможность просмотреть отправляемые данные.

Интерфейс IPostBackDataHandler определяет два метода:

- `LoadPostData()`. ASP.NET вызывает этот метод, когда страница отправляется назад, перед возбуждением любого события элемента управления. Он позволяет вам проверить данные, которые подлежат обратной отправке и соответствующим образом обновить состояние элемента. Однако вы не должны в этой точке инициировать события изменений, поскольку другие элементы могут быть еще не обновлены.
- `RaisePostDataChangedEvent()`. После того как все входные элементы управления на странице инициализированы, ASP.NET предоставляет вам шанс, если необходимо, инициировать событие изменения вызовом метода `RaisePostDataChangedEvent()`.

Разберём базовый пример. Элемент управления `CustomTextBox` эмулирует стандартный элемент `TextBox`. Он реализует интерфейс `IPostBackDataHandler`. Также `CustomTextBox` имеет свойство `Text`, которое сохраняется в состоянии вида. Генерация элемента производится в переопределённом методе `AddAttributesToRender()`. Вы должны добавить `UniqueID` к элементу управления, используя для этого атрибут `name`. Это связано с тем, что ASP.NET сверяет эту строку с отправленными данными. Если вы не добавите `UniqueID`, то метод `LoadPostData()` никогда не будет вызван, и вы не сможете извлечь отправленные данные.

```
using System;
using System.Collections.Specialized;
using System.Web.UI;
using System.Web.UI.WebControls;

public class CustomTextBox : WebControl, IPostBackDataHandler
{
    public CustomTextBox() : base(HtmlTextWriterTag.Input)
    {
        Text = string.Empty;
    }

    public string Text
    {
        get { return (string)ViewState["Text"]; }
        set { ViewState["Text"] = value; }
    }

    protected override void AddAttributesToRender(
        HtmlTextWriter output)
    {
        output.AddAttribute(HtmlTextWriterAttribute.Type, "text");
        output.AddAttribute(HtmlTextWriterAttribute.Value, Text);
        output.AddAttribute("name", UniqueID);
        base.AddAttributesToRender(output);
    }
```

```
    public bool LoadPostData(string postDataKey,
                             NameValueCollection postCollection)
    {
        // методы будут реализованы позже
    }

    public void RaisePostDataChangedEvent()
    {
        // методы будут реализованы позже
    }
}
```

Реализуем методы интерфейса `IPostBackDataHandler`. Метод `LoadPostData()` принимает два параметра. Первый параметр - ключ, идентифицирующий данные для текущего элемента управления. Второй параметр - коллекция значений, переданных странице. Таким образом, вы можете получить доступ к данным для вашего элемента управления, используя примерно такой синтаксис:

```
string newData = postCollection[postDataKey];
```

Метод `LoadPostData()` должен сообщить ASP.NET о том, требуется ли событие изменения. Если вы вернете `true`, ASP.NET вызовет метод `RaisePostDataChangedEvent()` после того, как все элементы управления будут инициализированы. Перед `RaisePostDataChangedEvent()` стоит относительно простая задача – запустить событие, определённое в пользовательском элементе управления.

```
public bool LoadPostData(string postDataKey,
                         NameValueCollection postCollection)
{
    var postedValue = postCollection[postDataKey];
    if (Text != postedValue)
    {
        Text = postedValue;
        return true;
    }
    return false;
}

public void RaisePostDataChangedEvent()
{
    OnTextChanged(new EventArgs());
}

public event EventHandler TextChanged;

protected virtual void OnTextChanged(EventArgs e)
{
```

```
        if (TextChanged != null)
        {
            TextChanged(this, e);
        }
    }
```