Министерство образования и науки Российской Федерации
Ярославский государственный университет им. П. Г. Демидова
Кафедра вычислительных и программных систем

В. В. Васильчиков

Программирование ASP.NET Web Forms

Учебно-методическое пособие

Ярославль
ЯрГУ
2021

УДК 004.42(075.8)
ББК 3973.2-018.1я73
В19

Рекомендовано
Редакционно-издательским советом университета
в качестве учебного издания. План 2021 года

Рецензент
кафедра вычислительных и программных систем
ЯрГУ им. П. Г. Демидова

Васильчиков, Владимир Васильевич.

В19 Программирование ASP.NET Web Forms : учебно-методическое пособие / В. В. Васильчиков ; Яросл. гос. ун-т им. П. Г. Демидова. — Ярославль : ЯрГУ, 2021. — 208 с.

В пособии рассмотрены вопросы создания Web-приложений с использованием технологии ASP.NET Web Forms и среды разработки Microsoft Visual Studio на языке C#.

Предназначено для студентов, изучающих дисциплину «Программирование ASP.NET».

Библиогр.: 6 назв.

УДК 004.42(075.8)
ББК 3973.2-018.1я73

Введение

Занятия по программированию на языке C# приложений и компонентов, предназначенных для работы в среде .NET Framework, на факультете информатики и вычислительной техники ЯрГУ автором проводились на протяжении нескольких последних лет. В качестве среды разработки традиционно используется Microsoft Visual Studio различных версий.

Литературы по разработке Windows- и Web-приложений на C# в .NET Framework в настоящее время довольно много, но она издается малыми тиражами и не всегда доступна, в том числе и по цене. Кроме того, объем изложенного там материала весьма велик, что не всегда удобно для первого знакомства с предметом.

По разным аспектам разработки Windows-приложений для .NET Framework ранее автором были изданы три учебных пособия [4–6]. Данное пособие посвящено разработке Web-приложений на C# в среде программирования Microsoft Visual Studio с использованием технологии ASP.NET Web Forms. Занятия по соответствующей учебной дисциплине проводятся уже несколько лет, и студентами неоднократно высказывалось пожелание иметь учебное издание с текстом лекций.

Автор исходит из предположения, что ранее студенты освоили курс по разработке приложений Windows-приложений для .NET Framework, а значит, знакомы с языком C# и средой разработки Microsoft Visual Studio.

Как обычно, для углубления знаний по данному предмету студентам рекомендуется воспользоваться книгами, изданными в сериях «для профессионалов». Некоторые издания названы в списке рекомендуемой литературы. Часть этих изданий к настоящему моменту имеет более свежие редакции, ориентированные на новые версии языка и среды .NET Framework. Автор не стал их указывать, поскольку в учебном процессе сейчас используются C# версии 4.0 и .NET Framework версии 4.0 или 4.6. Новшества же, появившиеся в более поздних версиях, носят довольно специальный характер и в рамках учебного процесса не рассматриваются.

Лекция 1. Разработка Microsoft ASP.NET Web-приложений в Microsoft Visual Studio

Microsoft Visual Studio позволяет вам разрабатывать мощные Web-приложения без необходимости ручного создания громадного количества кода. .NET Framework обеспечивает вас компонентами для создания многокомпонентных распределенных приложений. Одной из множества образующих .NET Framework технологий является Microsoft ASP.NET, которая и предназначена для разработки Web-приложений.

В рамках данной темы мы разберем ключевые черты .NET Framework и ASP.NET.

Раздел 1. Введение в .NET Framework

В этом разделе мы ознакомимся с терминологией и основными концепциями .NET Framework применительно кразработке Web-приложений, а также вкратце рассмотрим средства, предоставляемые средой Visual Studio для их создания.

.NET Framework — это инфраструктура Microsoft.NET для выполнения приложений Windows, приложений Web, других приложений и служб. Это относится и к серверным и к клиентским приложениям.

Перечислим преимущества использования .NET Framework для создания и эксплуатации Web-приложений.

- Поддержка существующих Internet-технологий.
- Унифицированная модель разработки и использования для всех .NET приложений, включая Windows- и Web-приложения.
- Использование общей для .NET системы типов, простота взаимодействия с компонентами, созданными на других языках программирования, поддерживаемых .NET.
- Удобство проектирования функциональности, использование для этого иерархии пространств имен и классов.
- Легкость развертывания на тестовом или основном сервере.
- Легкость внесения изменений в развернутое приложение с использованием службы Windows Update.

Перечислим некоторые технологии, с которыми нам предстоит работать при изучении данного учебного курса.

- ASP.NET. Это технология разработки Web-приложений и Web-служб.
- Технологии доступа к базам данных, которые в основном базируются на ADO.NET. Последняя включает в себя Entity Framework и Data Services.
- WCF (Windows Communication Foundation). Технология предлагает единую программную модель для разработки приложений, ориентированных на службы.

Раздел 2. Обзор ASP.NET

В рамках данного раздела мы

- рассмотрим процесс взаимодействия клиента и сервера;
- опишем технологию ASP.NET в целом, а также перечислим компоненты, на которых она основана;
- разберем, как генерируется разметка страницы и код, реализующий ее функциональность;
- познакомимся с моделью исполнения, основанной на динамической компиляции;
- узнаем, что такое расширения ASP.NET Framework.

Клиент-серверное взаимодействие

Отметим для начала, что разработка приложений с использованием протокола HTTP принципиально отличается от разработки Windows-приложений. Этот протокол по своей природе бесстатусный, то есть он не сохраняет состояние взаимодействующих сторон.

Сеанс взаимодействия клиента и сервера имеет примерно такой сценарий.

- Клиент запрашивает у Web-сервера страницу, сформировав URL-запрос. Сервер в ответ должен сформировать и отправить HTML-код, возможно с включением кода клиентской стороны.
- Соединение между клиентом и сервером устанавливается при запросе и разрывается после ответа сервера. Если требуется сохранять какую-либо информацию между

запросами, потребуется задействовать дополнительные механизмы, мы их рассмотрим позже.

- После получения страницы и отображения ее браузером клиент может с ней взаимодействовать, например, нажимать кнопки. Это приводит к обратной посылке (postback) с использованием методов POST или GET. В ответ на серверной стороне может быть выполнен некоторый код и клиенту будет послана обновленная страница (или ее часть).

Технология ASP.NET делает организацию такого взаимодействия весьма несложной для программиста. При этом используется модель, основанная на понятиях события и обработчика события.

Что собой представляет ASP.NET

Ниже мы перечислим основные характеристики технологии ASP.NET.

- Она представляет собой каркас для построения динамических Web-приложений.
- Это унифицированная Web-платформа, обеспечивающая нас службами и средствами для создания приложений профессионального уровня.
- Технология основана на использовании .NET Framework.
- Она независима от выбора языка программирования и используемых браузеров.
- Код серверной стороны пишется на объектно-ориентированном языке и основан на использовании Web Forms и механизма обработки событий.
- Технология ASP.NET обеспечивает поддержку сетевых WCF-служб.

Как только что было сказано, пользовательский интерфейс наших приложений будет основан на использовании Web Forms. Как правило, приложение имеет одну, а чаще несколько Web-форм. Также вы можете использовать JavaScript, на стороне сервера различные .NET-языки и технологии (например, jQuery, Ajax) для расширения функциональности своего приложения.

Приложения ASP.NET машинно-независимы, им требуется только наличие браузера на стороне клиента. Их удобно разрабатывать и тестировать в среде Microsoft Visual Studio.

Для проведения аутентификации нам предлагается несколько готовых сценариев. Чаще используется встроенная Windows-аутентификация или аутентификация, основанная на формах. После развертывания ASP.NET-приложение запускается как процесс под управлением IIS (Internet Information Services). Технология ASP.NET также обеспечивает поддержку WCF-служб. Они представляют собой распределенные приложения, использующие форматы XML и SOAP для передачи информации между службами и приложениями-клиентами.

Компоненты Web-приложения ASP.NET

Перечислим основные файлы, которые могут присутствовать в вашем приложении ASP.NET.

- Web-формы. Соответствующие файлы имеют расширение .aspx.
- Пользовательские контролы (User Controls). Файлы имеют расширение .ascx.
- Мастер-страницы (Master Pages). Файлы имеют расширение .master.
- Файлы конфигурации (.config). Это наши настройки для приложения в формате XML.
- Global.asax. Код для обработки событий уровня приложения, в том числе неперехваченных прерываний.
- Ссылки на используемые Web-службы.
- Файлы для взаимодействия с источниками данных. Они могут иметь самую разную природу.

Также в вашем приложении могут использоваться и различного рода дополнительные файлы: HTML для статического контента, текстовые файлы, данные в формате XML, XSLT-файлы для их преобразования в HTML, стилевые файлы (.css) и многие другие.

Формирование и рендеринг разметки и кода приложения

Для создания Web-приложения необходимо сформировать внешний вид страниц и код их поддержки. Все это должно

удовлетворять некоторым стандартам, чтобы браузер мог корректно отобразить вашу страницу и обработать действия пользователя с ней.

Для начала в самом общем виде опишем, что должно произойти с вашим готовым приложением прежде, чем его увидит пользователь. Отметим только, что программист может создать в Visual Studio два типа проектов: ASP.NET Web Site project и ASP.NET Web application.

О разнице между этими вариантами мы поговорим чуть позже, пока отметим, что следующее описание относится к варианту ASP.NET Web Site project.

- Сначала все необходимые файлы должны быть скопированы на Web-сервер, локальный или удаленный. Когда пользователь обращается за той или иной страницей, происходит разбор (парсинг) содержимого страницы и для ее кода вызывается подходящий компилятор. Полученный код MSIL далее преобразуется в машинный код Just-in-time (JIT) компилятором. Это делается и для встроенного (inline), и для фонового (code-behind) кода. Таким образом, Web-формы и пользовательские контролы превращаются в классы с методами для отображения соответствующих страниц или элементов управления.
- Результат компиляции (одна или несколько сборок) хранятся в папке для временных файлов ASP.NET. Запрошенная страница, разумеется, может быть любой, а не только домашней страницей или страницей по умолчанию. Соответствующая сборка располагается в подпапке с именем, производным от ее имени. Впоследствии разработчик в любой момент может внести изменения в Web-форму и ее код, они скажутся на работе приложения немедленно.

Отметим, что, хотя исходный код и хранится на Web-сервере, пользователи доступа к нему не имеют. Однако он доступен пользователям вашей сети, имеющим право доступа к этим папкам. Если такое положение дел вас не устраивает, вы можете выбрать вариант ASP.NET Web Application либо использовать предварительную компиляцию.

Динамическая модель компиляции ASP.NET

По умолчанию при запросе пользователем ресурса в первый раз происходит компиляция страницы и ее кода. Результат кэшируется, чтобы последующие запросы обрабатывались быстрее.

Разберем последовательность событий при первом запросе.

1. Браузер клиента обращается с HTTP-запросом к серверу.
2. На сервере происходит разбор (парсинг) исходного кода.
3. Если код еще не был скомпилирован в dll-сборку, вызывается компилятор.
4. Готовая сборка (на языке MISL) загружается в память и запускается на выполнение.

При втором и последующих запросах последовательность короче.

1. Браузер клиента обращается с HTTP-запросом к серверу.
2. Сборка, полученная ранее, немедленно загружается в память и запускается на выполнение.

Как было отмечено ранее, вы можете воспользоваться предварительной компиляцией своих страниц. В этом случае вы получаете перечисленные ниже преимущества.

- Скорость ответа на первый запрос возрастет, поскольку для него уже не требуется компиляция.
- Вы можете обнаружить ошибки в коде (синтаксические) еще до запроса пользователя.
- При развертывании приложения на сервере вы не выставляете исходный код.

Расширения ASP.NET Framework

В рамках данного учебного курса мы изучим два расширения ASP.NET Framework: ASP.NET Ajax и ASP.NET Dynamic Data.

ASP.NET Ajax (Asynchronous JavaScript and XML) — это скорее не технология, а комбинация стандартов, которая обеспечивает возможность лучше реализовать богатый потенциал функциональности современных Web-браузеров.

Ключевым средством Ajax-ориентированных Web-приложений является способность Web-браузера взаимодействовать с Web-сервером через операции, известные под названием

асинхронных или частично-страничных обратных отправок. Они позволяют обновить отдельные части страницы, не пересылая страницу целиком.

ASP.NET Ajax также позволяет автоматически создавать код клиентской стороны на языке JavaScript, чем могут воспользоваться даже разработчики, не знающие этого языка.

Разработчик также может воспользоваться богатым набором интерактивных элементов управления (для стороны клиента) Ajax Control Toolkit и обширной библиотекой jQuery с открытым исходным кодом на языке JavaScript.

ASP.NET Dynamic Data добавляет к существующим в ASP.NET контролам для работы с данными (data controls) основные возможности быстрой разработки приложений RAD (Rapid Application Development).

Это специальный каркас, который мы можем использовать для полнофункциональной работы с данными после подключения к их источнику. При этом обеспечивается поддержка всех основных операций: Create, Read, Update, Delete, а также автоматическая проверка на корректность, основанная на ограничениях базы данных на типы, длину полей, разрешение значения null и т. п.

ASP.NET Dynamic Data позволяет нам легко создавать управляемые данными Web-приложения ASP.NET. Во время работы автоматически задействуются метаданные используемой модели данных и на их основе выстраивается пользовательский интерфейс приложения. Данные легко просматриваются и редактируются. При этом нам предоставляется возможность внести изменения в поведение по умолчанию путем модификации тех или иных элементов или через создание новых. Перечисленные возможности не требуют обязательного создания нового приложения, их можно добавить в уже существующие.

Лекция 2. Создание Web-приложений в Microsoft Visual Studio на языке C#

Мы исходим из того, что студенты знакомы с языком программирования C# и интегрированной средой разработки Microsoft Visual Studio, поэтому позволим себе не останавливаться на описании их характеристик. В данной теме мы рассмотрим процесс создания простейшего ASP.NET Web-приложения в Microsoft Visual Studio и ознакомимся с двумя вариантами создания соответствующего проекта.

Раздел 1. Создание простого Web-приложения

Процесс разработки Web-приложения

Создание Web-приложения можно разделить на несколько этапов.

1. Разработка дизайна проекта
   1.1. Разработка структуры приложения. Разумеется, Visual Studio заметно облегчает эту работу, предоставляя соответствующие шаблоны и инструменты. Однако это не избавляет разработчика от необходимости ясно представлять требования заказчика и возможности будущего приложения. Если вам нравится визуальное проектирование, то входящий в состав Visual Studio инструмент Class Designer позволит вам наглядно представить структуру ваших классов и отношения между ними. Вы можете легко добавить в проект новый класс или изменить существующий.
   1.2. Создание исходного проекта. Вы можете для этого выбрать один из готовых шаблонов. Visual Studio автоматически создаст проект соответствующего типа, сгенерирует необходимые файлы и минимальный код поддержки. На этом этапе целесообразно отметить в Task List основные участки кода, который вам предстоит создать.
   1.3. Создание пользовательского интерфейса, написание кода. Для создания интерфейса очень удобно пользоваться окном редактора в режиме Design. После добавления элементов на Web-форму для их настройки можно

использовать окно Properties. Необходимый код, в том числе код реакции на события, удобно писать в окне редактора кода.

2. Отладка

   2.1. Собираем проект. При этом Visual Studio весь код Web-страниц и других классов превращает в сборку dll. Вы можете выбрать для компиляции вариант Debug или Release. Откомпилировать можно и отдельный проект, и решение целиком.

   2.2. Тестируем и отлаживаем проект, используя инструменты, предоставляемые средой Visual Studio.

3. Развертывание

   Когда отладка завершена, собираем все в режиме Release и развертываем необходимые файлы на Web-сервере

Файлы и папки проекта Web-приложения для варианта Web application project

В проект Web-приложения включается много файлов и папок для их размещения. Перечислим основные файлы и папки, входящие в проект Web application project с использованием языка C#.

1. Папка *App_Data*. Предназначена для хранения данных приложения, таких как файлы баз данных, структурированные данные в формате XML и многие другие.
2. Папка *Bin*. Содержит результирующие сборки проекта.
3. Папка *Obj*. Содержит промежуточные результаты компиляции.
4. Файл *Default.aspx*. Форма для вашей страницы по умолчанию.
5. Файл *Default.aspx.cs*. Фоновый код для вашей страницы по умолчанию.
6. Файл *Default.aspx.designer.cs*. Код частичного класса для работы со страницей по умолчанию в процессе ее разработки.
7. Файл *Web.config*. Файл с настройками конфигурации вашего Web-приложения.
8. Файл *WebApplicationName.csproj*. Файл проекта (текст в формате XML). Содержит ссылки на все элементы проекта,

такие как формы и классы, а также ссылки на прочие проекты и настройки процесса компиляции.

9. Файл *WebApplicationName.csproj.user*. Пользовательские настройки для проекта.

Файлы и папки проекта Web-приложения для варианта Web Site project

Перечислим основные файлы и папки, входящие в проект Web Site project, создаваемый на языке C#.

1. Папка *Account*. Предназначена для хранения данных, используемых для аутентификации.
2. Папка *App_Data*. Предназначена для хранения данных приложения, таких как файлы баз данных, структурированные данные в формате XML и многие другие.
3. Папка *Scripts*. Содержит обычно файлы библиотеки jQuery (исходный код на скриптовом языке), но может использоваться для хранения и пользовательских скриптов.
4. Файлы *About.aspx* и *About.aspx.cs*. Форма и соответствующий код для вашей странички About.
5. Файлы *Default.aspx* и *Default.aspx.cs*. Форма и соответствующий код для вашей страницы по умолчанию.
6. Файл *Global.asax*. Файл, содержащий множество методов, имеющих отношение к приложению в целом и к текущему сеансу работы с клиентом.
7. Файлы *Site.master* и *Site.master.cs*. Мастер-страница и ее фоновый код.
8. Файл *Web.config*. Файл с настройками конфигурации вашего Web-приложения.

Сравнение вариантов Web application project и Web Site project

При создании Web-приложения выбор между этими двумя вариантами определяется вашими предпочтениями. Одни разработчики предпочитают модель Web Site. Она проще, поскольку ресурсы определяются неявным образом за счет их размещения в тех или иных папках. Другие предпочитают вариант Web Application, потому что эта модель обеспечивает больший контроль над проектом, поскольку ресурсы явно

определяются в файле проекта. Собственно, в этом и состоит основное отличие.

При работе с Web Site вы можете изменить проект, добавив или убрав файлы в тех или иных папках. Если же вы имеете дело с вариантом Web Application, это надо делать в среде Visual Studio посредством команд add/remove.

Другим важным отличием является то, что при работе с вариантом Web Application в процессе компиляции и построения все классы проекта собираются в единую сборку, которая размещается в папке Bin. При развертывании приложения именно этот код, а не исходный вместе с файлами разметки страниц, пользовательских контролов и т. п. копируется на Web-сервер. При работе же с вариантом Web Site вы обычно выкладываете на сервер исходный код и используете динамическую компиляцию.

Также при работе с Web Site вы можете запускать на выполнение или отладку отдельную страницу, не собирая проект целиком. Это удобно, если на других страницах еще не исправлены все ошибки. В случае Web Application вам придется собрать весь проект.

Кроме использования динамической компиляции, преимущество Web Site заключается еще и в том, что после развертывания файлы можно добавлять или изменять. Впрочем, если вы не хотите делать доступным свой исходный код, вам придется использовать Web Application или Web Site с предварительной компиляцией.

Перечислим еще несколько особенностей этих двух вариантов, которые могут повлиять на ваш выбор.

Вы можете предпочесть Web Site, если

- вы хотите иметь отдельную сборку для каждой страницы;
- вы хотите открывать и редактировать директорию, как Web-проект, не создавая собственно проекта.

Вариант Web Application вы можете выбрать, если

- вы хотите контролировать имена окончательных сборок;
- в рамках одного решения хочется иметь несколько Web-проектов;

- требуется дополнительное управление процессом компиляции.

Основные файлы Web application project

Перечислим основные типы файлов, которые мы могли бы создать при разработке Web-приложения.

- ASP.NET Web-формы (.aspx). Это основной строительный материал для создания динамических Web-сайтов. Web-форма может быть запрошена пользователем непосредственно через ее URL. Функциональность Web-формы обычно задается файлом ее фонового кода с расширением .aspx.cs (при использовании C#).
- Службы WCF (.svc). Вы можете их подключить для использования функциональности, предоставляемой другими программами. Их ключевыми компонентами являются контракты или интерфейсы, а также файлы с фоновым кодом (.cs).
- Классы. Соответствующие файлы на языке C# имеют расширение .cs.
- Файлы JavaScript (.js). Они могут содержать код стороны клиента, написанный на языке JavaScript.
- Файл Global.asax — опциональный, он содержит код обработки событий уровня приложения. В период выполнения программы этот файл компилируется в динамически создаваемый класс, наследующий классу **HttpApplication**.
- Файлы мастер-страниц (.master). Мастер-страницы позволяют создавать постоянное (повторяющееся) наполнение страниц вашего Web-приложения: оформление и функциональность.
- Файлы ресурсов (.resx). Под ресурсами мы подразумеваем различного рода данные, связанные с вашим приложением и требующие копирования при развертывании последнего. Хранение данных в специальных файлах ресурсов позволяет изменять их без необходимости перекомпиляции приложения.

- Styles.css. Каскадные таблицы стилей предоставляют нам простой механизм добавления стилей оформления страниц (шрифты, цвета, выравнивание и т. п.).
- Файл Web.config. Файл содержит настройки конфигурации, которые используются средой исполнения, такие как политики связывания сборок, использование служб WCF. Также в файле могут содержаться различного рода настройки и переменные, используемые приложением.
- Файл Web.sitemap. Файл в формате XML содержит описание элементов навигационного характера: меню, т. н. «хлебные крошки».

Лекция 3. Создание ASP.NET Web-формы

Раздел 1. Создание Web-формы

Web-форма представляет собой комбинацию разметки, контролов (элементов управления) и кода, который исполняется на Web-сервере (например, IIS). Web-формы нередко называют страницами ASP.NET или .aspx-страницами. Их удобно создавать в Visual Studio и программировать на языке C# (или других языках, поддерживаемых средой разработки). При создании Web-формы вы можете выбирать, размещать ли код прямо на форме (в файле .aspx) или поместить его в отдельный файл с фоновым кодом (code behind).

В данном разделе мы ознакомимся с вопросами создания Web-форм и их основными характеристиками.

Как выглядит описание Web-формы?

С функциональной точки зрения Web-форма — это контейнер для текста и контролов, которые мы хотим отобразить в окне браузера на стороне клиента. Для этого она должна сгенерировать разметку на языке HTML и послать ее браузеру.

Однако Web-форма не является обычной Web-страницей, поскольку в отличие от последней содержит код серверной стороны, который срабатывает в ответ на определенные события. Браузеру посылается только сгенерированная разметка страницы и, возможно, код клиентской стороны на скриптовом языке. Такое разделение расширяет круг используемых браузеров, улучшает функциональность и безопасность Web-приложения.

Обычно Web-форма состоит из двух файлов: файла разметки (.aspx) и кода поддержки формы (.aspx.cs). Здесь и далее мы предполагаем, что в качестве языка программирования мы выбрали C#. В разметке Web-формы обычно присутствуют следующие пять элементов:

- директива Page и связанные с ней атрибуты, определяющие общую функциональность;
- элемент !DOCTYPE и связанные с ним атрибуты, определяющие соответствующий форме тип документа (DTD — document type definition);

- элемент HTML и связанные с ним атрибуты указывающие, что форма содержит код на языке гипертекстовой разметки HTML;
- элемент BODY и связанные с ним атрибуты, представляющие содержимое страницы;
- элемент FORM и связанные с ним атрибуты, определяющие как обрабатываются группы контролов, присутствующих на форме.

Ниже приводится для примера типичный вид директивы Page:

```
<%@ Page Title="" Language="C#" AutoEventWireup="true"
    CodeFile="Default.aspx.cs" Inherits="Default" %>
```

Перечислим атрибуты, которые может содержать эта директива.

- Атрибут AutoEventWireup указывает на факт автоматического возбуждения событий уровня страницы (например, Page_Load). Для реакции на событие следует определить метод-обработчик. По умолчанию значение этого атрибута при использовании языка C# равно true, для других языков это не обязательно так. Например, для Basic значение по умолчанию равно false.
- Для этих событий не требуется указывать имя обработчика, оно стандартное, например, Page_Load или Page_Init. Параметры соответствующего метода имеют такой же смысл, как и для всех прочих обработчиков событий.
- Атрибут CodeBehind (для варианта Web application project) задает имя файла, содержащего код класса, связанного со страницей.
- Атрибут CodeFile (для варианта Web Site project) также задает имя файла, объявляющего класс, связанный со страницей. В этом случае дополнительно используется атрибут Inherits.
- Атрибут Inherits определяет имя класса, от которого наследует страница. В приведенном примере это **Default**.
- Атрибут Language определяет, на каком языке написан код серверной стороны.

- Атрибут Title позволяет задать заголовок страницы, отображаемый в заголовке браузера (или вкладке). Его можно также указать и в элементе HTML. Если он указан и в Title, и в HTML, приоритетом пользуется значение из директивы Page.

Элемент HTML, как было сказано, означает, что форма содержит HTML-разметку. Внутри него может находиться элемент BODY, определяющий наполнение страницы:

```
<html>
    <body class="body" title="This page...">
    </body>
</html>
```

Атрибуты элемента BODY имеют следующий смысл:

- class определяет класс таблицы стилей (CSS), которая применяется к элементам на Web-форме;
- title — строка-подсказка, которая появляется при наведении курсора на HTML-элемент или на серверный контрол, размещенные на Web-форме.

Внутри элемента BODY обычно находится элемент FORM, его типичный вид:

```
<form id="form1" runat="server">
    ...
</form>
```

Атрибуты этого элемента:

- id — идентификатор, используемый кодом серверной стороны при необходимости программного доступа к форме;
- method определяет способ передачи данных о запросе. Значение post (по умолчанию) означает, что данные передаются в виде пары ключ/значение в теле HTTP-запроса, при указании значения get данные передаются в строке запроса;
- атрибут runat="server" означает, что форма обрабатывается на стороне сервера, куда и направляется информация из контролов страницы. Если этот атрибут опущен, то мы имеем дело с обычной страничкой HTML.

Раздел 2. Добавление серверных контролов на Web-форму и их настройка

После создания Web-формы необходимо сформировать ее наполнение. Обычно мы будем использовать для этого серверные контролы.

Что такое серверные контролы?

Серверные контролы — это элементы управления, они объединяют в себе пользовательский интерфейс и функциональность. Соответствующий код выполняется на стороне сервера, на что указывает обязательный для них атрибут runat="server". В качестве примеров серверных контролов можно назвать кнопки, поля ввода, выпадающие списки.

Серверные контролы могут сохранять свое состояние во время работы Web-приложения, для этого используется специальный скрытый контрол **ViewState**. Они обладают встроенной функциональностью, позволяющей запускать на стороне сервера код, который реагирует на воздействие пользователя на контрол. Этот код прописывается в соответствующем обработчике события. Основная часть функциональности приложения содержится именно в этих методах-обработчиках.

В ASP.NET серверные контролы используют общую модель функционирования, поэтому некоторые атрибуты совпадают для многих контролов. Например, атрибут BackColor для всех определяет цвет фона. Типичный вид серверного контрола (кнопки) в файле .aspx:

```
<asp:Button id="Button1" runat="server" BackColor="Red"
        Width="238px" Height="25px" Text="Web control" />
```

Для правильного отображения страницы браузером Web-серверный контрол определяет тип браузера и генерирует для него подходящий HTML-код. Например, если браузер поддерживает скрипты на клиентской стороне, контрол посылает скрипт для реализации своей функциональности. Если браузер скриптов не поддерживает, будет сгенерирован соответствующий код серверной стороны. В этом случае, конечно, потребуется много обращений к серверу, но функциональность реализована будет.

Пусть у нас на форме присутствует поле ввода, объявленное так:

```
<asp:TextBox id="UsernameTextBox" runat="server"
    Width="238px" Height="25px">Enter your Username
</asp:TextBox>
```

Для браузера Internet Explorer версии 8+ будет сгенерирован такой HTML-код:

```
<input name="UsernameTextBox" type="text"
    value="Enter your Username" id="UsernameTextBox"
    style="height:25px;width:238px;" />
```

Для других браузеров код, возможно, будет другим, но это не забота разработчика.

Типы серверных контролов

Есть два типа серверных контролов: HTML-серверные контролы и Web-серверные контролы.

HTML-серверные контролы

По умолчанию обычные HTML-элементы на Web-форме серверу недоступны. Они рассматриваются как плоский текст, обрабатываемый браузером. Но если к ним добавить атрибут runat="server", то они превращаются в HTML-серверные контролы и могут иметь код, выполняющийся на стороне сервера.

В качестве примера приведем некоторые часто используемые HTML-элементы:

- a — ссылка на другую страницу, изображение, иной ресурс:

```
<a href="/Page2.aspx" title="Go To Page 2.">Page 2</a>
```

- img — вставка изображения:

```
<img src="/login.png" alt="Log in to web application." />
```

- input — различные элементы ввода информации (тип определяется значением type):

```
<input type="text" name="UserName">
```

- select — элемент для выбора одного из предопределенных значений:

```
<select size="3" name="OptionList">
```

```
    <option selected value="Option1">
        Option 1</option>
    <option value="Option2">
        Option 2</option>
    <option value="Option3">
        Option 3</option>
</select>
```

Web-серверные контролы по их функциональному назначению можно разделить на несколько категорий, которые мы перечислим ниже.

Стандартные Web-серверные контролы

Контролы из этой категории можно разделить на встроенные и сложные контролы. Встроенные контролы зачастую соответствуют простым HTML-элементам, таким как кнопки, списки, поля ввода, и используются примерно так же, как и соответствующие HTML-элементы.

К сложным контролам обычно относят контролы для работы со статическими или динамическими данными, контейнеры для других контролов, различные многофункциональные контролы. Как пример можно привести элемент управления Календарь, который предоставляет удобный интерфейс для ввода даты в правильном формате.

В разметке встроенных Web-серверных контролов присутствует префикс "asp:", пример был выше. Перечислим некоторые часто используемые Web-серверные контролы из этой категории:

- *<asp:Button... />* — кнопка;
- *<asp:CheckBox... />* — поле для выбора опции;
- *<asp:HyperLink... />* — гиперссылка, аналог HTML-элемента anchor;
- *<asp:Image... />* — изображение;
- *<asp:ImageButton... />* — кнопка, помеченная изображением, а не текстом;
- *<asp:Label... />* — нередактируемый текст (метка);
- *<asp:ListBox... />* — список возможных вариантов для выбора, в том числе множественного;
- *<asp:Panel... />* — разделение формы на области (без границы), которые используются как контейнеры для других контролов;

- *<asp:RadioButton... />* — радио-кнопка;
- *<asp:Table... />* — таблица;
- *<asp:TextBox... />* — поле для ввода текста.

В качестве примера сложных Web-серверных контролов можно назвать следующие:

- *AdRotator* — демонстрация предопределенной или случайной последовательности изображений;
- *Calendar* — удобный (графический) ввод даты;
- *FileUpload* — контрол для загрузки файлов с сервера или на сервер;
- *Wizard* — мастер, обеспечивающий пользовательский интерфейс и навигацию, чтобы собрать информацию за несколько шагов.

Контролы для работы с данными

Контролы из этой категории (так называемые data-контролы) можно разделить на две группы: элементы для показа данных из некоторого источника (data-bound контролы) и промежуточные элементы для связи источников данных с другими data-контролами.

В качестве примера контролов из группы data-bound можно назвать следующие:

- *Chart* — контрол для удобного отображения статистической или финансовой информации (таблица, график, диаграмма и т. п.);
- *DataList* — показывает строки из базы данных в удобном (и настраиваемом) виде;
- *DetailsView* — отображает строку из источника данных так, что каждое поле выводится в отдельной строке;
- *FormView* — показывает строку из источника данных в соответствии с заданным шаблоном отображения;
- *GridView* — показывает данные в виде таблицы;
- *ListView* — отображает данные в соответствии с заданными шаблонами и стилями. Неявно поддерживает операции Edit, Insert, Delete, а также обеспечивает сортировку и разбиение на страницы. Это верно и для данных, не связанных ни с каким источником;

- *QueryExtender* — используется для фильтрации данных при получении их из источника;
- *Repeater* — отображает информацию из кэша (data set) в указанных вами HTML-элементах и контролах, делая это по одному разу для каждой записи в data set.

Приведем примеры обычно используемых data-контролов из второй группы (data source):

- *AccessDataSource* — привязывает данные из базы данных Access к работающим с ними контролам;
- *EntityDataSource* — позволяет работать с моделью доступа Entity Data Model (EDM);
- *LinqDataSource* — позволяет организовать взаимодействие с базой данных по технологии LINQ;
- *SiteMapDataSource* — представляет навигационные данные, полученные от поставщика карты сайта;
- *SqlDataSource* — позволяет подключиться к реляционному источнику данных, например Microsoft SQL Server;
- *XmlDataSource* — позволяет подключиться к данным в формате XML.

Контролы для проверки корректности ввода

Контролы из этой категории (контролы валидации) — это специальные скрытые элементы управления, чтобы реагировать на ситуацию, когда пользовательский ввод не удовлетворяет заданным условиям. Обычно мы в этом случае выводим соответствующее сообщение. Для осуществления такой проверки контрол валидации должен быть ассоциирован с контролом, осуществляющим ввод.

Приведем примеры обычно используемых контролов из этой категории:

- *CompareValidator* — проверяет соответствие пользовательского ввода вводу повторному или значению некоторого поля;
- *CustomValidator* — позволяет проверить корректность ввода программным путем (написав соответствующий метод);
- *RangeValidator* — проверяет попадание введенных данных в заданный диапазон;

- *RegularExpressionValidator* — проверяет соответствие ввода заданному регулярному выражению;
- *RequiredFieldValidator* — не позволяет полю ввода остаться пустым;
- *ValidationSummary* — собирает информацию от всех контролов валидации на данной странице и выводит сообщение о результатах проверки.

Контролы из категории Login Controls

ASP.NET предоставляет нам удобные контролы для работы с информацией об учетных записях пользователей. Это следующие элементы управления:

- *ChangePassword* — позволяет пользователю изменить свой пароль для данного Web-сайта;
- *CreateUserWizard* — позволяет зарегистрировать нового пользователя;
- *Login* — предоставляет стандартный интерфейс для проведения аутентификации;
- *LoginName* — отображает имя пользователя, прошедшего аутентификацию;
- *LoginStatus* — отображает ссылку, зависящую от того, прошел пользователь аутентификацию или нет;
- *LoginView* — отображает различную информацию для анонимных и аутентифицированных пользователей;
- *PasswordRecovery* — позволяет пользователю восстановить пароль для данного Web-сайта.

Контролы для навигации по Web-страницам

К этой категории относятся следующие элементы управления:

- *Menu* — показывает главное меню и меню следующего уровня, допускает динамическую настройку;
- *SiteMapPath* — показывает местоположение текущей страницы на карте сайта и позволяет быстро перемещаться в обратном направлении;
- *TreeView* — служит для отображения иерархических данных в виде дерева.

Сохранение состояния элементов управления

При разработке Web-приложения одним из первых возникает вопрос, как сохранять состояние контролов при пересылках между клиентом и сервером. Дело в том, что мы используем протокол HTTP, а он по своей природе бесстатусный.

ASP.NET предлагает использовать для этой цели специальный скрытый контрол __VIEWSTATE. Требующая пересылки информация хранится в нем в виде закодированных строк. При этом весь процесс кодирования и декодирования автоматизирован и программист не должен о нем заботиться, его забота — дизайн и функциональность страниц.

Включение и выключение режима сохранения состояния

По умолчанию состояние контролов сохраняется в элементе __VIEWSTATE. Однако если контролов на странице много, количество пересылаемой информации может стать слишком большим, что негативно скажется на скорости работы приложения. Поэтому нам предоставляется возможность включения и отключения режима сохранения состояния как для отдельных контролов, так и для страницы в целом.

Так, для отключения сохранения состояния для всей страницы мы можем в директиве Page использовать атрибут EnableViewState:

```
<%@ Page EnableViewState="false" %>
```

А это пример отключения сохранения состояния для отдельного контрола:

```
<asp:ListBox id="ListBox1" EnableViewState="false"
             runat="server" />
```

Для управления режимом сохранения состояния можно также использовать атрибут ViewStateMode. Приведем пример отключения сохранения состояния для всей страницы и его включения для отдельного контрола:

```
<%@ Page ViewStateMode="Disabled" %>
...
<asp:ListBox id="ListBox1" " runat="server"
             ViewStateMode="Enabled" />
```

Атрибут ViewStateMode может принимать следующие значения:

- *Inherit* — значение свойства ViewStateMode наследуется от родительского контрола или страницы;
- *Enabled* — режим сохранения состояния включен, даже если он был отключен для родительского контрола;
- *Disabled* — режим сохранения состояния отключен, даже если он был включен для родительского контрола.

Отметим, что не следует путать понятие режима сохранения состояния (View state) и состояние контрола (Control state). Первое позволяет помнить его текущее состояние, например выбранный элемент в выпадающем списке, второе — основа его поведения. Например, есть контролы, которые заново заполняются каждый раз при обращении за страницей, скажем, таблицы со значениями из базы данных. Для них мы можем отключить View state, но не Control state.

Для хранения и передачи информации можно также использовать Web-серверный контрол **HiddenField**. У него есть свойство Value, а на сервере мы можем использовать обработчик события ValueChanged, предназначенный для выполнения каких-либо действий, если значение Value изменилось. Ниже приводится пример использования этого контрола в разметке страницы:

```
<asp:HiddenField id="HiddenField1" runat="server"
                 value="1"/>
```

Добавление и настройка HTML-серверных контролов и Web-серверных контролов

Для добавления на страницу HTML-серверных контролов в среде разработки Microsoft Visual Studio их можно просто перетащить с панели инструментов (Toolbox). Для дальнейшей их настройки удобнее всего далее воспользоваться окном свойств (Properties). Также, если на странице имеются обычные элементы HTML, они легко превращаются в HTML-серверные контролы путем добавления к ним атрибута runat="server".

В качестве возможных преимуществ использования этой категории контролов можно отметить следующие:

- легкое превращение имеющейся HTML-страницы в Web-форму;
- возможность определить, какие элементы работают локально в браузере, а какие на сервере, что могло бы позволить оптимизировать производительность нашего приложения.

Добавление и настройка Web-серверных контролов удобнее всего выполняется в Visual Studio таким же образом: с использованием панели Toolbox и окна Properties.

Выбор между HTML-серверными контролами и Web-серверными контролами

При разработке Web-приложения у вас могут быть различные аргументы в пользу выбора между этими двумя категориями серверных контролов.

Вы могли бы выбрать использование HTML-серверных контролов, если:

- вы привыкли к стилю программирования HTML-страниц, код будет практически таким же;
- у вас есть готовая HTML-страница и вам надо быстро добавить для нее функциональность, как у Web-формы;
- пропускная способность вашего канала ограничена и вы хотели бы поменьше всего делать на стороне сервера.

В то же время существует множество Web-серверных контролов, не имеющих аналогов в HTML. Также в пользу выбора этого варианта могли лечь следующие соображения. К примеру, вы предпочитаете объектно-ориентированный стиль программирования, вам удобно различать контролы по их идентификаторам (ID), а также иметь отдельно пользовательский интерфейс и отдельно код.

Кроме того, предпочесть Web-серверные контролы вы могли бы по следующим причинам.

- С ними вы легко можете создавать Web-приложения, пригодные для показа практически любым из браузеров. Кроме того, код, исполняемый на сервере, обладает существенно бóльшими возможностями, чем скриптовый код на стороне клиента.

- Вам нужна специфическая функциональность, предоставляемая сложными Web-серверными контролами, например **Calendar** или **AdRotator**.
- Вас не слишком сдерживают производительность вашего сервера и пропускная способность канала связи.

Лекция 4. Реализация функциональности ASP.NET Web-формы

Раздел 1. Работа с фоновым кодом

Существует несколько вариантов размещения кода, обеспечивающего функциональность Web-страницы. Можно использовать, например, встроенный или смешанный код, который размещается непосредственно на странице (в файле .aspx). Однако наиболее предпочтительный вариант — использование фонового кода (code-behind). В Visual Studio именно он является вариантом по умолчанию.

Основу этого кода составляют обработчики событий разного уровня. Они срабатывают при загрузке страницы или в ответ на какое-либо действие пользователя. Чтобы научиться правильно использовать этот механизм, нам потребуется в том числе разобраться с жизненным циклом страницы. Это необходимо и при разработке собственных контролов для того, чтобы мы могли в нужный момент инициализировать контрол или получить доступ к его свойствам.

Весь код, с которым мы здесь будем иметь дело, — это код серверной стороны, он срабатывает перед тем, как страница будет отправлена клиенту. Это верно и в тех случаях, когда некоторые контролы автоматически создают код клиентской стороны.

В рамках данной темы мы разберем разные методы добавления кода в Web-приложение ASP.NET. В последнем разделе мы научимся использовать события уровня страницы, в особенности событие OnLoad (метод-обработчик Page_Load).

Варианты размещения кода

Перечислим допустимые варианты размещения кода серверной стороны.

- *Смешанный код (Mixed code).* Код размещается в том же файле, что и разметка страницы (.aspx) и перемешан с ним.
- *Встроенный код (Inline code).* Код размещается в том же файле, что и разметка страницы, но в отдельной секции.

- *Фоновый код (Code-behind).* Код размещается в отдельном файле, в Visual Studio это вариант по умолчанию.

Смешанный и встроенный код

Оба варианта допустимы, однако не рекомендуются к использованию, особенно первый, когда код перемешан с разметкой страницы.

Встроенный код выглядит несколько аккуратнее, поскольку помещен в отдельную секцию файла .aspx, ниже приводится пример:

```
<html xmlns="http://www.w3.org/1999/xhtml">
...
<asp:Button ID="Button1" runat="server"
            OnClick="Button1_Click"
            Text="Button" />
...
</html>
<script Language="C#" runat="server">
    private void Button1_Click(object sender,
                               System.EventArgs e)
    {
    ...
    }
</script>
```

Фоновый код

Это самый предпочтительный вариант, он используется по умолчанию. Разметка страницы содержится в файле с расширением .aspx, а соответствующий код в файле с таким же именем и расширением .aspx.cs (при использовании языка C#). При создании приложения можно использовать и разные языки, однако в одном файле может присутствовать код только на одном языке.

Обычно файл с фоновым кодом содержит единственный класс с именем, совпадающим с именем соответствующей Web-формы, например:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
```

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

public partial class Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        ...
    }
}
```

Здесь метод Page_Load — это заготовка обработчика события Page.OnLoad.

Использование файлов с фоновым кодом

Перечислим основные моменты использования этих файлов. Для правильной работы Web-приложения необходимо, чтобы каждая .aspx-страница ассоциировалась с файлом, содержащим фоновый код. Разумеется, этот код потребуется скомпилировать. И тут допускаются два варианта:

- компиляция на лету (Just-in-Time — JIT), которая происходит при первом запросе страницы, — это вариант по умолчанию;
- предварительная компиляция приложения перед его размещением на сервере.

Для настройки процесса компиляции в директиве Page предусмотрен ряд атрибутов.

- *CodeBehind*. Указывается имя файла с кодом. Атрибут используется для варианта Web application project.
- *CodeFile*. То же самое, но для варианта Web Site project.
- *Inherits*. Позволяет .aspx-странице наследовать от класса, прописанного в файле с фоновым кодом. Отметим, что при использовании языка C# этот атрибут чувствителен к регистру.

Также в дополнение к этим атрибутам может использоваться атрибут AutoEventWireup, который указывает, будут ли события для Web-формы генерироваться автоматически. В случае использования языка C#, значение этого атрибута по умолчанию

равно true. Дизайнер автоматически генерирует код для привязки событий к их обработчикам.

Напомним, что Visual Studio поддерживает два типа Web-приложений: Web application project (WAP) и Web Site project (WSP). Кроме всего прочего, они различаются и режимом компиляции.

- WAP. Компиляция происходит по команде Build. Скомпилированный код на языке Microsoft intermediate language (MSIL) помещается в единую сборку проекта (обычно в подпапке Bin). При первом запросе сборка перемещается во временную папку ASP.NET, где она компилируется окончательно в машинный код посредством JIT-компилятора. Это приводит к небольшой задержке удовлетворения первого запроса по сравнению с последующими.
- *WSP*. По умолчанию предварительная компиляция не выполняется. Она происходит при первом запросе страницы. Такое поведение еще называют компиляцией на месте (in place). Приложение компилируется в одну или несколько сборок во временной папке ASP.NET при первом запросе. Размещение приложения на сервере довольно простое. Кроме того, вы можете вносить изменение в уже развернутое приложение. Также у вас остается возможность осуществить предварительную компиляцию, чтобы не выкладывать исходный код на сервер. Для этого есть опция Publish Web Site в меню Build.

Раздел 2. Обработка событий серверных контролов

Существуют два типа обработчиков событий: обработчики клиентской стороны и обработчики серверной стороны. Последние требуют отправки информации на сервер, соответственно, срабатывают медленнее, но предоставляют больше возможностей, в частности, доступ к ресурсам сервера, например к базе данных.

Что такое обработчики событий?

События — основной механизм, позволяющий нашему приложению реагировать на действия пользователя. Информация

о воздействии на тот или иной элемент управления отправляется на сервер, и там запускается код обработчика соответствующего события. Ниже приводится пример разметки, содержащей привязку обработчика события OnClick для кнопки на форме и пример кода этого обработчика с XML-комментариями. Он перенаправляет пользователя на страницу Default.aspx.

```
<html>
...
<body>
    <form runat="server">
    ...
    <asp:Button ID="Button1" Text="Home"
            OnClick="Button1_Click" runat="server"/>
    ...
    </form>
</body>
</html>

/// <summary>
/// Перенаправление на домашнюю страницу
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
protected void Button1_Click(object sender,
                             EventArgs e)
{
    // Перенаправление на домашнюю страницу
    Response.Redirect("~/Default.aspx");
}
```

Что такое обработчики клиентской стороны?

Вообще-то обработчики клиентской стороны изначально предназначались для элементов HTML. Их можно прицепить и к серверным контролам, но только к тем, которые при отображении превращаются в один или несколько HTML-элементов. Ниже приводится пример кода такого обработчика на языке JavaScript. Он изменяет цвет элемента при прохождении над ним курсора мыши. Также приводится код привязки обработчика к событию.

```
...
<script type="text/JavaScript">
    function changeColor()
    {
       window.event.srcElement.style.color = "#FF0000";
    }
</script>
...
<asp:Button id="Button1" runat="server" text="Button1"
            onmouseover="changeColor();" />
```

Это так называемая статическая привязка, однако осуществить ее можно и в коде серверной стороны во время выполнения приложения:

```
Button1.Attributes.Add("onmouseover",
                       "changeColor();");
```

Впрочем, и сам код функции-обработчика можно добавлять динамически. Эта функция с кодом на JavaScript должна быть частью разметки страницы. Она может там присутствовать явно, как в приведенном примере, или ее можно поместить на страницу через регистрацию в коде серверной стороны. Соответствующий код можно поместить в какой-нибудь обработчик события, например события PageLoad:

```
ClientScriptManager buttonClientScriptManager =
                                        this.ClientScript;
buttonClientScriptManager.RegisterClientScriptBlock(
    Button1.GetType(),"changeColorScript",
    "function changeColor() {
       window.event.srcElement.style.color = '#FF0000';
    }",
    true
);
```

Здесь мы получаем ссылку на класс **ClientScriptManager** через ссылку на свойство ClientScript объекта **Page**. После этого мы регистрируем функцию, приводя полностью ее код. Однако, несмотря даже на такую регистрацию, код клиентской стороны не может иметь доступа к ресурсам сервера, например базе данных.

Обработчики клиентской стороны полезны, если событие нужно обработать немедленно, поскольку для них отсутствует

необходимость передачи информации на сервер и обратно. Они широко используются для предварительной проверки введенной пользователем информации перед посылкой ее на сервер.

Что такое обработчики серверной стороны?

В отличие от клиентских обработчиков для обработчиков серверной стороны требуется отсылать информацию на сервер для обработки. Скорость реакции при этом, разумеется, уменьшается, зато возможности намного больше, чем в случае работы на клиентской стороне, поскольку мы в этом случае имеем доступ к ресурсам сервера.

Код обработчиков серверной стороны располагается на сервере. Они могут использоваться и с Web-, и с HTML-серверными контролами. Ранее упоминалось, что код серверной стороны в том числе может содержаться непосредственно на странице .aspx. Ниже приводится шаблон для оформления в этом случае скриптовой секции:

```
<script type="text/C#" runat="server">...</script>
```

Множество событий, которые можно обрабатывать на серверной стороне, ограничено. На стороне клиента вы можете обрабатывать, например, события мыши и событие onChange. На серверной стороне поддерживаются события Click и специальный вариант события onChange, а вот обрабатывать на сервере события мыши было бы глупо.

Создать обработчик события серверной стороны в среде Visual Studio очень просто. Для создания обработчика для события по умолчанию достаточно выполнить двойной клик на контроле, для остальных событий выделяем контрол и выбираем в окне Properties необходимое событие на вкладке Events. Среда создает шаблонную заготовку обработчика и осуществляет привязку к элементу в файле .aspx и сгенерированном ею коде поддержки формы.

Использование свойств Web-серверных контролов

Для программного доступа к серверным контролам они обязательно снабжаются уникальным идентификатором, в разметке представленным как атрибут ID. Значение этого атрибута в коде серверной стороны используется как имя переменной для доступа к соответствующему объекту, свойства которого в раз-

метке представлены также в виде атрибутов. Приведем простой пример разметки формы с двумя элементами управления:

```
<form id="form1" runat="server">
    <asp:TextBox ID="NameTextBox" runat="server" />
    <asp:Button ID="SubmitButton" runat="server" />
</form>
```

Доступ к свойствам этих контролов осуществляется так же, как и к свойствам любых объектов в языке C# на чтение:

```
string greetingString = "Hello " + NameTextBox.Text;
```

и на запись:

```
GreetingLabel.Text = "new text";
```

Раздел 3. Обработка событий уровня страницы

Жизненный цикл Web-страницы включает целый ряд шагов, таких как:

- инициализация страницы и контролов;
- создание контролов;
- сохранение и поддержка состояния;
- запуск обработчиков событий;
- показ страницы (рендеринг) и многое другое.

Ниже мы разберем, как использовать события страницы, в том числе при обслуживании обратных посылок (postback).

События жизненного цикла страницы

Когда Web-страница запрошена, происходит целый ряд событий, в частности: инициализация, загрузка страницы, валидация (проверка корректности), обработка событий обратной посылки, рендеринг, выгрузка страницы и т. п. Для выполнения необходимых действий в ответ на то или иное событие, необходимо их прописать в коде соответствующего обработчика.

Имена обработчиков для событий уровня страницы заданы заранее; события, к которым они привязаны, ясны из имен обработчиков. Приведем несколько примеров.

- *Page_Init*. Обработчик события, означающего, что все элементы управления на странице инициализированы.

- *Page_Load*. Обработчик события, означающего, что все элементы управления на странице загружены. Событие происходит после *Page_Init*.
- *Page_LoadComplete*. Событие происходит после *Page_Load*. В этот момент можно использовать либо задать значения контролов.
- *Page_Unload*. Обработчик события, означающего, что страница закрыта или мы перешли на другую страницу.

Кроме событий уровня страницы, требуется обрабатывать события, относящиеся к элементам управления на странице. Имена соответствующих обработчиков программист может задать по своему желанию; Visual Studio предлагает при создании обработчиков имена, состоящие из имени контрола и имени события, например Textbox1_Changed или Button1_Click.

Процесс обратной посылки

При обработке событий нередко требуется знать, запрашивается ли страница в первый или не в первый раз. Чаще всего такая необходимость возникает при написании кода обработчика Page_Load, который срабатывает при любом запросе страницы. При первом запросе там нередко в контролы загружаются начальные значения, в остальных случаях этого обычно делать не нужно, но зато нужно знать состояние контролов после воздействия на них пользователя.

Понять, с какой ситуацией мы имеем дело, позволяет булевское свойство IsPostBack объекта **Page**. Оно равно истине, если страница запрашивается не в первый раз. Пример кода обработчика:

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!Page.IsPostBack)
    {
        // Действия при первом запросе страницы
    }
    // Действия при любом запросе
    ...
}
```

Процесс обратной посылки по умолчанию происходит только при нажатии кнопок на форме. Изменение, скажем, текста в поле

ввода или выборе элемента из выпадающего списка к обратной посылке не приводит. В некоторых случаях такое поведение страницы требуется изменить. Можно сделать это путем установки атрибута AutoPostBack для заданного контрола в значение true. В этом случае любые события на нем будут приводить к обратной посылке.

Лекция 5. Создание и использование мастер-страниц и пользовательских контролов

Мастер-страницы предоставляют разработчику средство для удобного включения в Web-формы своего приложения повторяющегося содержимого, включая и пользовательский интерфейс, и код.

Примерно для этой же цели предназначены пользовательские контролы. Обычно вы создаете их как комбинацию других контролов, создаете для них необходимый код и оформляете все как единый компонент, который можно добавлять к различным страницам.

Раздел 1. Мастер-страницы

Мастер-страницы можно рассматривать как некий слой ASP.NET-приложения, оформленный в файле с расширением .master. Они как бы накладываются на другие страницы. Если вы внесете изменения в мастер-страницу, они немедленно будут отражены во всех страницах, на ней основанных.

Что такое мастер-страница?

Мастер-страница — это своего рода охватывающий слой приложения, содержащий общий контент для многих или даже всех страниц. На них можно размещать и статический текст, и HTML-элементы и серверные контролы. ЗНАКИ

Оформляются они в файлах с расширением .master. Вместо директивы Page на них имеется директива Master. В одном приложении может быть несколько мастер-страниц. К примеру, одна используется для пользователей, прошедших аутентификацию, другая — для анонимных пользователей.

Типичная директива Master выглядит так:

```
<%@ Master Language="C#" AutoEventWireup="true"
           CodeFile="MasterPage.master.cs"
           Inherits="MasterPage" %>
```

Отметим здесь, что Web-серверы, например IIS, не позволяют просматривать в браузере файлы с расширением .master из соображений безопасности.

Вместе с тем мастер-страница — это почти стандартная Web-форма с обычным наполнением. Основная ее особенность — это наличие одного или нескольких контролов **ContentPlaceHolder**, которые определяют место для подстановки содержимого из собственно страницы. Размещаться они могут, например, внутри элемента head или form:

```
<html>
<head runat="server">
   <title>General Application Title</title>
   <asp:ContentPlaceHolder id="HeadContentPlaceHolder"
                           runat="server">
   </asp:ContentPlaceHolder>
</head>
<body>
   <form id="form1" runat="server">
      ...
      <! Стандартное наполнение >
      <div class="top">
         <asp:ContentPlaceHolder
         id="MainContentPlaceHolder" runat="server" />
         ...
         <! Стандартное наполнение >
      </div>
      <div class="bottom">
         <asp:ContentPlaceHolder
            id="FooterContentPlaceHolder"
            runat="server">
            <asp:Label id="FooterLabel" runat="server"
                Text="Footer Text" />
         </asp:ContentPlaceHolder>
      </div>
      ...
   </form>
</body>
```

Если страница, основанная на данной мастер-странице, не имеет контрола **Content**, который ссылается на контрол **ContentPlaceHolder**, результирующая страница будет показана как есть, то есть так:

```
<div class="bottom">
      <asp:ContentPlaceHolder
```

```
            id="FooterContentPlaceHolder"
            runat="server">
            <asp:Label id="FooterLabel" runat="server"
               Text="Footer Text" />
        </asp:ContentPlaceHolder>
</div>
```

Создать заготовку мастер-страницы удобнее всего с использованием мастеров среды Visual Studio. Делается это так же, как и для обычной страницы, через диалоговое окно Add New Item, только там требуется выбрать вариант Master Page. После этого мы можем наполнять ее содержимым.

Что такое контент-страница?

Это страницы, которые наполняют содержимым мастер-страницы. Они имеют расширение .aspx и могут запрашиваться браузером. При создании контент-страницы нужно указать мастер-страницу, на которой она будет основана, впрочем, это можно сделать и потом, добавив атрибут MasterPageFile в ее директиву Page:

```
<%@ Page Language="C#"
   MasterPageFile="~/MasterPage.master"
   AutoEventWireup="true"CodeFile="ContentPage.aspx.cs"
   Inherits="ContentPage" %>
```

Также можно в файле конфигурации Web.config указать мастер-страницу для всего приложения:

```
<pages masterPageFile="MasterPage.master" />
```

Контент-страница должна содержать как минимум один контрол **Content**, который ссылается на соответствующий **ContentPlaceHolder** на мастер-странице:

```
<asp:Content ID="MainContent"
     ContentPlaceHolderID="MainContentPlaceHolder"
     runat="server">
</asp:Content>
<asp:Content ID="FooterContent"
     ContentPlaceHolderID="FooterContentPlaceHolder"
     runat="server">
</asp:Content>
```

Контрол **Content** обязательно должен содержать такую ссылку. Если ссылки нет или если контрол ссылается на несуществующий элемент **ContentPlaceHolder**, мы получим ошибку компиляции.

На контент-странице любое содержимое вне контролов **Content** (за исключением скриптовых блоков для серверного кода) также приводит к ошибке компиляции.

Через код контент-страницы вы можете делать что угодно, например, генерировать наполнение контрола **Content** или выполнять запросы к базе данных.

Создать заготовку контент-страницы в среде Visual Studio можно через диалоговое окно Add New Item, при создании можно указать необходимую для нее мастер-страницу.

Если на контент-странице есть ссылка на **ContentPlaceHolder**, в него вставляется содержимое контрола **Content**, если нет, то остается содержимое **ContentPlaceHolder** по умолчанию (если оно есть). Атрибут title из директивы Page также сливается с мастер-странице (замещает ее значение этого атрибута) при условии, что у мастер-страницы элемент head имеет атрибут runat="server".

Что такое вложенная мастер-страница?

На контент-страницу добавить контрол **ContentPlaceHolder** нельзя, однако мы можем использовать вложенные мастер-страницы. Например, родительская мастер-страница определяет внешний (охватывающий) слой, а дочерние мастер-страницы определяют слой отдельных областей на данном сайте. Директива Master для родительской страницы может выглядеть так:

```
<%@ Master Language="C#" AutoEventWireup="true"
           CodeFile="ParentMasterPage.master.cs"
           Inherits="ParentMasterPage" %>
```

а директива для дочерней страницы так:

```
<%@ Master Language="C#" AutoEventWireup="true"
           MasterPageFile="~/ParentMasterPage.master"
           CodeFile="ChildMasterPage.master.cs"
           Inherits="ChildMasterPage" %>
```

И расширение, и содержимое у дочерней мастер-страницы аналогичны родительской. Разумеется, она может иметь свои элементы **ContentPlaceHolder**.

При разработке портала или Web-сайта со множеством различных областей хорошей практикой является использование общей родительской и множества дочерних мастер-страниц.

Приведем пример родительской и дочерней мастер-страницы и контент-страницы, основанной на них. Родительская мастер-страница:

```
<% @ Master Language="C#" AutoEventWireup="true"
   CodeFile="ParentMasterPage.master.cs"
   Inherits="ParentMasterPage" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
   "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html >
<head runat="server">
   <title>Untitled Page</title>
</head>
<body>
<form id="Form1" runat="server">
   <div>
      <h1>Parent Master</h1>
      <p style="font:color=red">
         This is parent master content.</p>
      <asp:ContentPlaceHolder
         ID="MainContent" runat="server" />
   </div>
</form>
</body>
</html>
```

дочерняя мастер-страница:

```
<%@ Master Language="C#"
MasterPageFile="~/ParentMasterPage.master"%>
<asp:Content id="Content1"
   ContentPlaceholderID="MainContent" runat="server">
   <asp:Panel runat="server" id="PanelMain"
              backcolor="lightyellow">
      <h2>Child master</h2>
      <asp:Panel runat="server" id="Panel1"
                 backcolor="lightblue">
```

```
        <p>This is child master content.</p>
        <asp:ContentPlaceHolder ID="ChildContent1"
           runat="server" />
      </asp:Panel>
      <asp:Panel runat="server" id="Panel2"
                 backcolor="pink">
        <p>This is child master content.</p>
        <asp:ContentPlaceHolder ID="ChildContent2"
             runat="server" />
      </asp:Panel>
      <br />
   </asp:Panel>
</asp:Content>
```

контент-страница:

```
<%@ Page Language="C#"
MasterPageFile="~/Child.master"%>
<asp:Content id="Content1"
   ContentPlaceholderID="ChildContent1" runat="server">
   <asp:Label runat="server" id="Label1"
        text="Child label1" font-bold="true" />
   <br>
</asp:Content>
<asp:Content id="Content2"
   ContentPlaceholderID="ChildContent2" runat="server">
   <asp:Label runat="server" id="Label2"
        text="Child label2" font-bold="true"/>
</asp:Content>
```

Как обрабатывается мастер-страница?

При работе с мастер-страницами и контент-страницами вы можете использовать ряд событий, таких как Init или Load. Следует знать, в каком порядке они возникают. Когда пользователь запрашивает страницу (.aspx), ASP.NET проверяет директиву Page на предмет наличия ссылки на мастер-страницу. Разберем последовательность выполняемых при этом шагов:

1. Браузер клиента обращается с HTTP-запросом к серверу;
2. ASP.NET разбирает директиву Page, находит там атрибут MasterPageFile и подгружает мастер-страницу. При первом запросе и контент-страница, и мастер-страница проходят компиляцию;

3. ASP.NET объединяет содержимое мастер-страницы и контент-страницы;
4. При этом содержимое отдельных контролов **Content** помещается на место соответствующих элементов **ContentPlaceHolder**;
5. Результат отсылается браузеру.

С точки зрения программиста и контент-страница, и мастер-страница выступают как отдельные контейнеры для контролов. Контент-страница при этом может иметь доступ к открытым (public) членам мастер-страницы из своего кода. Например, вы можете хранить какой-нибудь объект в переменной состояния сеанса и сделать его доступным для всех контент-страниц через открытое свойство. Такой прием, в частности, позволяет избежать дублирования кода.

Также вам может потребоваться иметь программный доступ к некоторым контролам, например **SiteMapPath**, **Menu** или **Login**. Работать с ними также можно из кода контент-страницы. С этой точки зрения мастер-страница может рассматриваться и как дочерний элемент контент-страницы или как контейнер внутри контент-страницы (аналогично пользовательским контролам). Отличие ее от пользовательского контрола состоит в том, что мастер-страница является контейнером для *всех* серверных контролов, отображаемых браузером.

Добавление мастер-страницы к существующемуWeb-проекту

Если у вас есть ранее созданное Web-приложение, вы при желании можете добавить к нему одну или несколько мастер-страниц. Контент-страницы вы можете создать новые либо передать в них ранее созданные Web-формы.

Для такого преобразования потребуется выполнить следующие действия:

- создать мастер-страницу или добавить существующую;
- удалить из Web-формы все HTML-элементы верхнего уровня: html, head, body и form. Если в элементе head у вас был элемент title, переместите его в head мастер-страницы, убедившись, что там присутствует атрибут runat="server";

- прописать в директиве Page атрибут MasterPageFile со ссылкой на мастер-страницу;
- создать необходимое количество контролов **Content**, ссылающихся на контролы **ContentPlaceHolder** мастер-страницы;
- переместить существующие контролы в контролы **Content**;
- продумать, как перенести используемые стили непосредственно на мастер-страницу либо в используемый ею файл стилей (.css);
- расположить серверные контролы должным образом внутри контролов **Content**.

Резюме

Попробуем перечислить преимущества, которые вы получаете от использования механизма мастер-страниц при разработке Web-сайтов.

- Поддерживать Web-сайт легче. Изменения в одной мастер-странице немедленно отражаются во многих (или всех) страницах сайта.
- Исключается повторное использование разметки и кода.
- Становится удобнее проектировать, в каком виде представлять контент пользователю.
- Появляется возможность удобного разрешения или запрета редактирования (на уровне **ContentPlaceHolder**'ов).
- Появляется возможность использования одних и тех же мастер-страниц в разных Web-приложениях.
- Удобно работать с заголовками и подвалами (footer).
- Появляется возможность создания нескольких уровней вложенности (вложенные мастер-страницы).

Раздел 2. Пользовательские контролы

По своей природе пользовательские контролы — это страницы ASP.NET, которые Web-форма может импортировать как обычный серверный контрол. Пользовательский контрол, как и обычный, предоставляет разметку интерфейса и функциональность.

После создания пользовательского контрола его может использовать любая страница Web-приложения. Таким образом, упрощается повторное использование интерфейса и кода.

Варианты создания пользовательских контролов

Обычно используются следующие два варианта:

- можно расширить функциональность существующего серверного контрола, например, добавить к календарю текстовое поле, чтобы он загружал туда выбранную дату;
- можно для создания пользовательского контрола скомбинировать существующие элементы управления, чтобы они взаимодействовали между собой.

Для хранения пользовательских контролов на диске используются файлы с расширением .ascx. При этом, как обычно, предпочтительным вариантом хранения кода, реализующего логику контрола, являются файлы code-behind.

Так же, как и любая Web-форма, пользовательские контролы компилируются при первом запросе, а затем результат сохраняется в памяти сервера для обслуживания последующих запросов. Но, в отличие от обычных Web-форм, пользовательские контролы не могут быть запрошены сами по себе — только в составе страницы.

Основные черты пользовательских контролов

Они, как и Web-форма, содержат разметку HTML и код, но не содержат элементов html, body и form. Когда Web-форма использует пользовательский контрол, тот разделяет ее жизненный цикл, да и набор событий у них практически одинаковый.

Поскольку пользовательский контрол — это страница ASP.NET, у него есть собственная логика. Например, у него есть собственный обработчик события Page_Load.

Главная директива — Control (у обычной Web-форм, как мы помним, Page). Если используется фоновый код, то имя соответствующего файла указывается в данной директиве:

```
<%@ Control Language="C#" AutoEventWireup="true"
    CodeFile="WebUserControl.ascx.cs"
    Inherits="WebUserControl" %>
```

Директива Control поддерживает большинство атрибутов, поддерживаемых директивой Page, но не все. Например, она не поддерживает атрибут Trace. Он определяется для страницы, содержащей пользовательский контрол.

Перечислим основные черты, которыми отличаются друг от друга компоненты, Web-серверные контролы и пользовательские контролы.

- *Компонент* — это фактически библиотека классов, он обеспечивает только логику и не имеет пользовательского интерфейса.
- *Web-серверный контрол* — это традиционный элемент управления, например кнопка или поле ввода. Они обычно предоставляются средой разработки, однако вы можете создать и собственный. Они могут иметь пользовательский интерфейс, но могут и не иметь.
- *Пользовательский контрол* обязательно должен иметь пользовательский интерфейс и в момент разработки приложения, и, разумеется, во время его исполнения. Вы можете предоставить свой код для реализации его функциональности.

Преимущества и недостатки использования пользовательских контролов

Перечислим основные преимущества, который вы получаете при использовании пользовательских контролов.

- Они самодостаточны. Кроме того, они размещаются в отдельном пространстве имен, что исключает возможность конфликтов с методами и свойствами host-страницы.
- Их можно использовать многократно, в том числе в пределах одной host-страницы, конфликтов не будет.
- Их можно писать на языке, отличном от того, на котором создана host-страница.
- Один пользовательский контрол можно использовать на многих страницах Web-приложения.

Перечислим также те их черты, которые вы могли бы счесть недостатками.

- Совместное использование пользовательских контролов разными приложениями предполагает дублирование и интерфейса, и кода. Если потребуется внести изменения, вам придется заниматься всеми экземплярами данного контрола.
- При развертывании Web-приложения код виден, если только вы не используете предварительную компиляцию. Впрочем, это замечание относится и ко всему остальному коду.

Замечания о совместном использовании пользовательских контролов

Вы без проблем можете использовать один пользовательский контрол на всех страницах одного Web-приложения. Но если вы хотите использовать его в другом приложении, то вы должны будете его скопировать в виртуальную корневую папку этого Web-приложения.

Если нужен контрол для использования во многих Web-приложениях, можно создать заказной (Web custom control). Он функционирует как разделяемый пользовательский контрол, его можно даже добавить на панель инструментов среды Microsoft Visual Studio (Toolbox).

Есть только одна проблема — его нельзя создать средствами Microsoft Visual Studio, все придется писать вручную. Соответствующий класс должен быть наследником **Control** или **WebControl**.

Преобразование Web-формы в пользовательский контрол

Даже если вы изначально не думали об использовании пользовательских контролов в своем Web-приложении, вы можете их добавить в любой момент. Зачастую это даже лучше отложить до момента, когда станет понятно, какие элементы и код у вас дублируются.

Web-формы очень похожи на пользовательские контролы, поэтому их легко переделать. Для этого потребуется выполнить следующие действия:

- удалить элементы разметки html, head, body и form;

- переделать директиву Page в директиву Control, при этом удалив все атрибуты, за исключением Language, AutoEventWireup (если она есть), CodeFile и Inherits;
- добавить атрибут ClassName, который делает контрол строго типизированным при добавлении на host-страницу;
- изменить расширение файла .aspx на .ascx; если присутствует фоновый код, изменить расширение файла и для него;
- в фоновом коде изменить базовый класс с **System.Web.UI.Page** на **System.Web.UI.UserControl**;
- добавить специальные свойства для контролируемого доступа к закрытым переменным, если таковые имеются.

Добавление пользовательского контрола на Web-форму

Для использования пользовательского контрола на Web-форме, туда нужно поместить директиву Register, которая выглядит примерно так:

```
<%@ Register src="WebUserControl.ascx"
    tagname="WebUserControl" tagprefix="uc1" %>
```

Атрибуты этой директивы имеют следующий смысл:

- *tagprefix* определяет уникальное пространство имен, чтобы избежать конфликтов имен с другими пользовательскими контролами;
- *tagname* задает уникальное имя пользовательского контрола;
- *src* задает путь к файлу с пользовательским контролом.

После регистрации пользовательский контрол можно размещать на форме с идентификацией по имени:

```
<uc1:WebUserControl ID="WebUserControl1" runat="server" />
```

Замечание. В Visual Studio это все делается очень просто: достаточно контрол перетащить из окна Solution Explorer на форму. При этом среда разработки добавит директиву Register автоматически.

Когда запрашивается страница, содержащая пользовательский контрол, он также компилируется, что делает его доступным (программно) на странице.

Если пользовательский контрол имеет открытые свойства, следует обеспечить их правильную инициализацию. Это можно сделать:

- непосредственно в разметке;
- в окне Properties;
- в коде, примерно так:

```
WebUserControl1.Name = "Santa Claus";
```

Приведем еще один пример:

```
<%@ Page Language="C#" %>
<%@ Register TagPrefix="uc" TagName="Spinner"
    Src="~\Controls\Spinner.ascx" %>
<html>
    <body>
    <form runat="server">
        <uc:Spinner id="Spinner1" runat="server"
            MinValue="1" MaxValue="10" />
    </form>
    </body>
```

Здесь контрол содержится в файле Spinner.ascx в папке Controls. При регистрации указывается префикс uc и имя Spinner. При размещении задается id для программного доступа и инициализируются свойства MinValue и MaxValue.

Лекция 6. Проверка достоверности пользовательского ввода

Когда вы добавляете на форму контрол для ввода информации, например **TextBox**, вы делаете определенные предположения относительно того, что пользователь может ввести. Для того чтобы проверить, удовлетворяет ли его ввод вашим ожиданиям, и нужен процесс валидации.

ASP.NET предоставляет вам специальные контролы для такой проверки, а вы связываете с ними свои контролы ввода. Изначально эти действия выполняются на стороне клиента, чтобы не посылать на сервер заведомо неправильные данные.

Разумеется, проверять следует не только ввод пользователя, но и информацию, полученную из других источников, например Web-служб или внешних файлов. Поэтому, собственно, проверка и осуществляется на разных уровнях, но окончательная проверка, конечно, происходит на стороне сервера.

Если на странице есть несколько полей ввода, требующих проверки, то может потребоваться контрол, который проверяет, все ли они были заполнены правильно. Такой элемент управления также предоставляется нам средой разработки, и далее мы с ним познакомимся.

Раздел 1. Обзор процесса проверки достоверности пользовательского ввода

Цель данного раздела — разобраться, как работает процесс валидации (до того, как на сервер будет отправлен запрос). Еще раз напомним, что на стороне клиента производится предварительная проверка, а на стороне сервера окончательная.

Что такое валидация?

Проверка ввода делает работу пользователя более удобной: сообщения об ошибках появляются немедленно (хорошо бы они были достаточно информативными), причем сервер в этом даже не участвует. Также уменьшается риск обрушения Web-сайта из-за заведомо некорректного ввода.

Рассмотрим основные виды проверок и пользу, которую вы можете от них получить.

- Вы можете проверить ввод на соответствие заранее определенному формату. Это может быть:
  - количество вводимых символов;
  - использование букв и цифр;
  - допустимый диапазон значений;
  - специальный формат строки (например, адрес e-mail) или математической формулы.

  Это все делается специальными контролами. В случае ошибки они выводят сообщение и требуют повторить ввод.
- Предотвращается отправка данных на сервер (и обработка там) в случае, если они не прошли такую предварительную проверку. Даже если пользователь отключил скрипты и страница все же была отправлена на сервер, там (например, в обработчике события Page Load) вы можете проверить ее правильность, вызвав метод Page.Validate.
- Такой механизм позволяет бороться с двумя основными опасностями:
  - *Spoofing.* Это ситуация, когда пользователь подправляет HTML-страницу перед отправкой на сервер, как будто все проверки прошли успешно. Если мы осуществляем проверки и на серверной стороне, такой фокус уже не пройдет.
  - *Вредоносный код.* Если позволять пользователю вводить что угодно, то теоретически он может сформировать деструктивный код для сервера и его соединений, например:
    - устроить переполнение буфера и крах сервера;
    - изменить свои привилегии;
    - создать новый аккаунт;
    - создать запрос к базе данных и несанкционированно получить информацию.

Проверки клиентской и серверной стороны

Как уже говорилось ранее, в ASP.NET проверку можно проводить на обеих сторонах, причем на сервере обязательно, на стороне клиента опционально. На клиентской стороне она

поддерживается с помощью JavaScript либо других скриптовых языков. На стороне сервера используется любой из языков .NET, допускающий компиляцию в MSIL. Программная модель в обоих случаях используется по сути одна и та же, различие только в используемых языках и том, что с ними связано.

Проверка на стороне клиента улучшает качество использования Web-формы и уменьшает трафик. Нужно иметь в виду, что здесь присутствует некоторая специфика для разных браузеров (если они вообще поддерживают валидацию). Например, в древнем IE4 она происходила только по нажатию кнопки **Submit**, а в более свежих версиях можно перемещаться по контролам, используя мышь или кнопку **Tab**, при этом валидация выполняется немедленно. Но в любом случае все это только предварительная проверка.

Полная проверка осуществляется уже на сервере, это объясняется соображениями безопасности — вдруг что-то подменили, отключили проверочные скрипты и т. п. Для этого используется уже один из .NET-языков. Кроме того, на серверной стороне можно осуществить сравнение пользовательского ввода с сохраненными данными: паролями, географическими ограничениями (законы, тарифы) и т. д.

Раздел 2. Контролы проверки

Вообще-то проверка правильности пользовательского ввода — довольно нудный и трудоемкий процесс. Для каждого контрола нужно определить требования к вводу, написать код проверки для стороны сервера. Если вы хотите осуществлять еще и проверку на стороне клиента, потребуется написать аналогичный код на одном из скриптовых языков.

ASP.NET заметно облегчает эту работу, предоставляя нам набор специальных контролов и связанный с ними код проверки на стороне клиента и на стороне сервера.

Обзор контролов валидации

Для начала перечислим контролы из этого стандартного набора и узнаем, какого рода проверки мы можем выполнять с их помощью.

RequiredFieldValidator не позволяет оставить поле ввода пустым. Следует отметить, что все остальные валидаторы воспринимают пустой ввод как правильный.

CompareValidator осуществляет сравнение со значением другого контрола, с фиксированным значением, со значением, полученным из файла; может осуществлять проверку на совпадение типа. Примером его использования может служить сравнение пароля и его подтверждения.

RangeValidator проверяет попадание значения в заданный диапазон, немного напоминает CompareValidator.

RegularExpressionValidator проверяет введенное значение на соответствие заданному шаблону: e-mail, номер телефона, почтовый код и многое другое. Visual Studio предлагает нам несколько шаблонов, однако куда больше можно без труда найти в сети Интернет.

CustomValidator позволяет вам использовать для проверки собственный код. Например, вы можете проверить, является ли число простым или сравнить его со значением из базы данных.

ValidationSummary обеспечивает немедленное формирование состояния ввода для всей страницы, своего рода резюме. Обычно его размещают рядом с кнопкой **Submit**.

Основные контролы валидации и их свойства

Разберем основные типы проверок, выполняемых контролами валидации, и их свойства для необходимых настроек. У всех контролов есть свойство ControlToValidate, задающее идентификатор контрола ввода, значение которого следует проверить. Остальные свойства зависят от типа проверки.

RequiredFieldValidator

Используется для обязательного заполнения поля ввода, например учетной записи пользователя. Обычно связанный с ним контрол ввода инициализируется пустым значением. Если вы хотите использовать непустое инициализирующее значение (чтобы пользователь его заменил), то можно использовать свойство InitialValue:

```
<asp:TextBox id="NameTextBox" runat="server"
     Text="Enter your name" />
<asp:RequiredFieldValidator
```

```
id="NameRequiredFieldValidator" runat="server"
ControlToValidate="NameTextBox"
InitialValue="Enter your name"
ErrorMessage="You must enter your name"
Text="*" />
```

В этом случае значение, записанное в свойстве InitialValue, также будет недопустимым.

Это поле может оставаться пустым или содержащим начальное значение, пока вы работаете с другими полями страницы. Сообщение об ошибке появится тогда, когда вы переключите фокус ввода из этого поля на какой-нибудь другой контрол.

CompareValidator

Он используется для проверки на совпадение с заданным значением или со значением, введенным через другой контрол (например, пароль). Напомним, что пустой ввод рассматривается как правильный.

Для задания контрольного значения можно использовать свойства ValueToCompare или ControlToCompare, их смысл очевиден из названия. Если эти свойства заданы оба, то предпочтение отдается последнему.

Свойство Type позволяет перед сравнением привести значение к правильному типу. Это важно, поскольку результат сравнения зависит от типа сравниваемых значений. Допустимые значения этого свойства: String, Integer, Double, Date, Currency. Они соответствуют системным типам **System.String**, **System.Int32**, **System.Double**, **System.DateTime**, **System.Decimal** соответственно.

Свойство Operator задает операцию для проведения сравнения. Допустимые значения этого свойства: Equal, NotEqual, GreaterThan, GreaterThanEqual, LessThan, LessThanEqual и DataTypeCheck. Их смысл достаточно очевиден.

В качестве примера приведем разметку для сравнения на равенство значений из двух полей ввода:

```
<asp:TextBox id="PasswordTextBox" runat="server"
     Text="Enter your password" TextMode="Password" />
<br />
<asp:TextBox id="PasswordConfirmationTextBox"
     runat="server" Text="Confirm your password"
     TextMode="Password" />
```

```
<asp:CompareValidator id="PasswordCompareValidator"
     runat="server"
     ErrorMessage="The entered passwords do not match."
     ControlToCompare="PasswordTextBox"
     ControlToValidate="PasswordConfirmationTextBox"
     Text="*" />
```

RangeValidator

Контрол используется для проверки попадания значения в заданный диапазон. Его основные свойства: MinimumValue, MaximumValue, смысл которых очевиден, а также свойство Type, значения которого задаются так же, как и для **CompareValidator**.

В качестве примера рассмотрим проверку попадания возраста в диапазон от 18 до 50 лет:

```
<asp:TextBox id="AgeTextBox" runat="server"
     Text="Enter your age" />
<asp:RangeValidator id="AgeRangeValidator"
     runat="server" ControlToValidate="AgeTextBox"
     Type="Integer" MinimumValue="18" MaximumValue="50"
     ErrorMessage=
      "Applicants must be between 18 and 50 inclusive."
     Text="*" />
```

RegularExpressionValidator

Контрол осуществляет проверку на соответствие ввода некоторому предопределенному формату: телефонному номеру, почтовому коду, адресу электронной почты, адресу ресурса в глобальной сети, номеру карты социального страхования и многому другому. При проверке сравниваются образцы использования букв, цифр, спецсимволов и т. п.

Для задания шаблона используются так называемые регулярные выражения, написанные на своего рода языке, специально разработанном для таких конструкций. В Microsoft Visual Studio есть много образцов, доступных через пункт Validation Expression в окне Properties для данного контрола. Можно, конечно, написать и свое выражение, изучив соответствующий синтаксис, однако в глобальной сети очень легко найти готовое выражение практически для любого применения.

Приведем пример проверки на соответствие формату адреса электронной почты (до предела упрощенного):

```
<asp:TextBox id="EmailTextBox" runat="server" />
<asp:RegularExpressionValidator
     id="EmailRegexValidator" runat="server"
     ControlToValidate="EmailTextBox"
     ErrorMessage=
          "Use the format username@organization.xxx"
     ValidationExpression="\w+@\w+\.\w+" Text="*" />
```

Отметим, что при рендеринге страницы для браузера будет сгенерирован на языке JavaScript код, позволяющий осуществить проверку на стороне клиента. Здесь его приводить, по-видимому, нецелесообразно, однако при желании вы можете в окне браузера увидеть полученный им код.

CustomValidator

Контрол используется для реализации собственной логики проверки пользовательского ввода. Вы можете сравнить введенное значение со значением переменной, вычисленным значением, вводом из другого источника (скажем, пароль из базы данных), использовать абсолютно любой алгоритм проверки.

Такая проверка обычно осуществляется на серверной стороне, хотя при желании можно написать и код, работающий на стороне клиента (если позволяет браузер и все данные доступны). Однако в любом варианте этот весь код придется создавать вручную. Для стандартных же контролов валидации его нам предоставляла среда разработки.

У контрола **CustomValidator** есть два свойства для представления методов проверки на клиентской и серверной стороне: ClientValidationFunction и OnServerValidate. Первое задает функцию на скриптовом языке, выполняющую проверку на стороне клиента, второе — метод, делающий то же самое на сервере. Пример оформления разметки и кусочек кода серверной стороны (проверка числа на четность):

```
<asp:CustomValidator ID="NumberCustomValidator"
     runat="server"
     ClientValidationFunction="ClientValidationHandler"
     OnServerValidate="ServerValidationHandler"
     ControlToValidate="NumberTextBox"
     ErrorMessage="The number must be even" />

<script type="text/JavaScript">
```

```
function ClientValidationHandler(source, args)
{
     args.IsValid = (args.Value % 2 == 0)
}
</script>

protected void ServerValidationHandler(object source,
                              ServerValidateEventArgs args)
{
     args.IsValid = (args.Value % 2 == 0)
}
```

У обеих функций есть два аргумента, имеющих одинаковый смысл. Результат проверки возвращается через второй аргумент. У него есть два свойства: Value (через него получаем проверяемое значение) и IsValid (булевское, равно true, если проверка завершилась успехом).

У контрола **CustomValidator** свойство ControlToValidate обязательно заполнять не требуется, если оно не установлено, то функции в качестве проверяемого значения получают пустую строку.

Использование нескольких контролов валидации

Одного контрола валидации или одной функции проверки зачастую оказывается недостаточно, тогда мы можем использовать несколько.

Чаще всего вместе с контролом **RequiredFieldValidator** используют какие-то еще. Рассмотрим, к примеру, ввод номера телефона. Нам надо:

- чтобы пользователь указал номер,
- чтобы он соответствовал шаблону телефонного номера,
- чтобы он присутствовал в нашей базе данных.

Для первой проверки мы используем **RequiredFieldValidator**, для второй — **RegularExpressionValidator**, а для третьей нам придется задействовать **CustomValidator**, написав соответствующий код серверной стороны. Допустим, в разметке страницы поле для ввода номера задано так:

```
<asp:TextBox id="PhoneTextBox" runat="server" />
```

А так могла бы выглядеть разметка для необходимых трех валидаторов:

```
<asp:RequiredFieldValidator
     id="PhoneRequiredFieldValidator" runat="server"
     ErrorMessage="The telephone number is required."
     ControlToValidate="PhoneTextBox" Text="*" />

<asp:RegularExpressionValidator
    id="PhoneRegularExpressionValidator" runat="server"
    ErrorMessage="The telephone number is not correct."
    ControlToValidate="PhoneTextBox"
    ValidationExpression=
     "((\(\d{3}\) ?)|(\d{3}-))?\d{3}-\d{4}" Text="*" />

<asp:CustomValidator id="PhoneCustomValidator"
    OnServerValidate="PhoneServerValidationHandler"
    runat="server"
    ErrorMessage=
        "This telephone number is not recognized"
    ControlToValidate="PhoneTextBox" Text="*" />
```

Позиционирование и настройка контролов валидации на Web-форме

Расположение контролов валидации на Web-форме определяет, где будут появляться сообщения об ошибках. Удобно, если сообщение появляется рядом с контролом ввода, поэтому валидаторы обычно размещают сразу за элементом, к которому они относятся. Сообщения об ошибках задаются двумя атрибутами: ErrorMessage и Text.

ErrorMessage определяет сообщение, которое выводится на месте контрола валидации, если ввод неверный, а свойство Text не установлено. Это же сообщение включается в набор сообщений контрола **ValidationSummary**, если таковой на форме имеется (независимо от наличия свойства Text).

Text — альтернативное сообщение. Оно показывается на месте контрола валидации при неверном вводе.

Обычно, если на форме есть **ValidationSummary**, он используется для показа ErrorMessage, а справа от контрола выводят звездочку (Text = "*"), примерно так:

```
<asp:ValidatorType id="ID of validator" runat="server"
     ControlToValidate="ID of control"
     ErrorMessage="Error message for error summary"
```

```
    Display="Static | Dynamic | None"
    Text="Text to display next to the input control">
</asp:ValidatorType>
```

Свойство Display определяет, как будут расположены сообщения от разных валидаторов, если для данного контрола их задано несколько. Возможные варианты:

- *Static* — каждое сообщение выводиться на месте своего валидатора, даже если предыдущий не сработал;
- *Dynamic* — сообщения сдвигаются влево, чтобы не было просветов;
- *None* — вывод сообщения на месте валидаторов заблокирован.

Контрол **ValidationSummary** разумно расположить так, чтобы клиент видел его перед отправкой страницы на сервер. Поэтому его обычно располагают рядом с кнопкой **Submit** или ее аналогом.

Раздел 3. Проверка Web-форм

Контрол ValidationSummary

Контрол **ValidationSummary** осуществляет вывод, если свойство Page.IsValid установлено в значение false. В этом случае собираются и выводятся все сообщения ErrorMessage на данной странице.

Этот вывод может быть настроен по-разному: в виде окна или как текстовая область на странице, возможно с заголовком. В зависимости от установки свойства DisplayMode он может выглядеть как список или как единый параграф. Пример вывода сообщений в виде списка с заголовком:

```
<asp:ValidationSummary id="MyValidationSummary"
runat="server"
HeaderText="These errors were found:"
ShowSummary="True"
DisplayMode="List" />
```

Как мы уже сказали ранее, **ValidationSummary** обычно располагают рядом с кнопкой **Submit** или ее аналогом. При наличии этого контрола свойство Text обычно используют для пометки звездочкой полей ввода с некорректными значениями.

Программная валидация Web-форм

Вызывает интерес вопрос, в каком порядке срабатывает код серверной стороны, имеющий отношение к процессу валидации.

Контролы валидации выполняют проверки автоматически, когда Web-форма посылается обратно на сервер. Это происходит до того, как начнет работать код обработки событий. Однако при желании вы можете в нужный момент произвести валидацию самостоятельно программным путем. Вот примеры ситуаций, когда такая возможность могла бы быть полезной:

- вы добавляете контролы динамически во время исполнения программы;
- вы устанавливаете динамически некоторые свойства, например MinimumValue или MaximumValue для контрола **RangeValidator**;
- вы хотите проверить правильность страницы в обработчике события перед тем, как обращаться к ресурсам сервера, например к базе данных.

Свойство страницы IsValid имеет значение true, если все валидаторы расценили ввод как правильный. Однако оно установлено еще до того, как сработали какие-либо обработчики событий, в том числе и для контролов. Поэтому, если вам требуется это значение после обработки каких-то событий, следует вызвать метод Validate для страницы или отдельного контрола.

Ниже приводится пример программной валидации, когда вы используете контрол **RangeValidator**, а значение MinimumValue устанавливается во время работы, например, из конфигурационного файла. После запуска валидации вы можете проверить свойство IsValid для страницы. Если оно равно false, то можно пройтись по контролам валидации и понять, кто из них диагностировал ошибку.

```
// Обратная посылка?
if (this.IsPostBack)
{
// Запускаем валидацию страницы
    this.Validate();
```

```
    // Страница правильная?
    if (!this.IsValid)
    {
        // Определяем в цикле, кто из контролов
        // диагностировал ошибку
        foreach (IValidator controlValidator
                                        in this.Validators)
        {
            if (!controlValidator.IsValid)
            {
                ...
            }
        }
    }
}
```

Лекция 7. Средства отладки Web-приложений ASP.NET

Для целей отладки и трассировки нам предоставляются два удобных объекта: **Debug** и **Trace**. Именно в них мы будем записывать отладочную информацию.

Чем же они различаются? В первую очередь тем, трассировка возможна всегда, а для использования объекта **Debug** приложение нужно компилировать и запускать в отладочном режиме.

Раздел 1. Отладка в ASP.NET

Для начала отметим тот факт, что обычно отладка Web-приложения осуществляется локально. Microsoft Visual Studio допускает и удаленную отладку при условии, что на компьютере, откуда отладка происходит, установлены соответствующие программные инструменты. Этот вариант мы подробно рассматривать не будем.

Типы ошибок

Можно выделить три типа ошибок.

- *Синтаксические.* Они обнаруживаются и исправляются на этапе компиляции. Кроме того, редактор кода Visual Studio предупреждает о потенциальных ошибках еще на этапе проектирования.
- *Ошибки времени выполнения.* Основными средствами для их диагностики являются исключения и блоки try/catch.
- *Семантические ошибки.* Это логические ошибки, их труднее всего обнаруживать и исправлять. Именно для поиска таких ошибок и предназначены средства отладки, которые мы здесь рассматриваем.

Что мы понимаем под отладкой?

С точки зрения разработчика процесс отладки состоит из следующих трех шагов.

- Нужно зафиксировать сам факт наличия ошибки в логике приложения.

- Нужно найти причину ошибки и место, где она возникает.
- Нужно ошибку исправить.

Среда разработки Visual Studio предоставляет нам много удобных инструментов отладки: пошаговое выполнение, точки остановки, отслеживание значений переменных, возможность их изменить, просмотр состояния стека и много другое. Однако для поиска логических ошибок трудно обойтись без такого старого и проверенного средства, как отладочный вывод. Именно этот прием лежит в основе рассматриваемых ниже средств отладки.

Класс Debug

Класс **Debug** из пространства имен **System.Diagnostics** обеспечивает нас методами и свойствами для удобного вывода отладочной информации и осуществления различных проверок. Его использование не сказывается на производительности и размере вашего приложения, поскольку при компиляции в режиме Release он просто удаляется из кода.

Он предоставляет нам перечисленные ниже методы.

- Write. Запись строки в специальный объект из коллекции Listeners.
- WriteLine. То же, что и Write, в строку добавляется символ перехода на новую строку.
- WriteIf. То же, что и Write, при выполнении заданного условия.
- Print. Вывод форматированной строки текста.
- Assert. Вывод сообщения, если указанное условие не выполняется. В соответствующем диалоговом окне можно также увидеть стек вызовов.

Как собирается отладочная информация

Для сбора отладочной информации во время выполнения могут использоваться классы **Debug** и **TraceContext**. Они могут собирать и отображать информацию.

Для использования класса **Debug** надо скомпилировать приложение в отладочном режиме и запустить на выполнение

под управлением отладчика. При компиляции в режиме Release из кода ничего убирать не требуется, но класс **Debug** в сборку не включается. Методы этого класса направляют информацию текущей коллекции так называемых собирателей трассы (trace listeners). Те по умолчанию отображают ее в панели **Debug** окна Output.

Класс **TraceContext** позволяет либо отображать информацию прямо на Web-странице, либо накапливать ее в памяти, чтобы впоследствии использовать специальный просмотрщик. Класс доступен через свойство Trace текущей страницы.

Замечание. Не следует путать свойство Trace с классом **Trace** из пространства имен **System.Diagnostics**.

Методы для сбора отладочной информации

Назначение методов Write и WriteLine достаточно очевидно. У них, как и у других методов класса **Debug,** существует несколько перегрузок:

```
Debug.Write("Writing a string of text...");
Debug.WriteLine("Writing a line of text...");
Debug.Write("Writing a string of text...", "Category 1");
Debug.WriteLine("Writing a line of text...", "Category 2");
```

Здесь второй аргумент позволяет указать категорию, в которую следует поместить сообщение для удобства группировки.

Методы для вывода сообщения при выполнении некоторого условия (первый аргумент равен true) также имеют несколько перегрузок:

```
Debug.WriteIf(Page.IsPostBack,
          "Writing a string of text...");
Debug.WriteIf(Page.IsPostBack,
          "Writing a string of text...", "Category 1");
Debug.WriteLineIf(Page.IsPostBack,
          "Writing a line of text...");
Debug.WriteLineIf(Page.IsPostBack,
          "Writing a line of text...", "Category 2");
```

Метод Print выводит строку с форматированием:

```
Debug.Print("Printing a line of text...");
Debug.Print("Printing a line a text, Postback = {0}",
             Page.IsPostBack);
```

Метод Assert имеет несколько перегрузок, смысл аргументов понятен из примеров:

```
Debug.Assert(Page.IsPostBack)
Debug.Assert(Page.IsPostBack, "Brief Message")
Debug.Assert(Page.IsPostBack, "Brief Message",
             "Detail description message");
Debug.Assert(Page.IsPostBack, "Brief Message",
             "Detail description message",
             objectArrayToFormat);
```

Диалоговое окно с сообщением выводится в случае, если первый аргумент имеет значение false. В первом примере это диалоговое окно содержит только информацию о состоянии стека.

Локальная и удаленная отладка

Для начала вам следует определиться, будете ли вы вообще выводить отладочную информацию, использовать точки останова и т. п. Далее вы выполняете следующие действия.

- Разрешаете выполнение приложения в режиме отладки. Для этого в файле Web.config в нужном месте следует установить атрибут debug:

```
<configuration>
    <system.web>
        <compilation debug="true">
        ...
    </system.web>
<configuration>
```

- Добавляете код для вывода отладочной информации, точки останова.
- Запускаете приложение в отладочном режиме.
- Следите за исполнением по шагам или точкам останова, используя выведенную отладочную информацию и окна со значениями переменных, состоянием стека и т. п.

Удаленная отладка позволяет вам с одной рабочей станции отлаживать Web-приложения на разных серверах. Это может потребоваться, если у вас нет возможности запустить приложение локально либо вы хотите протестировать ранее развернутое

где-то приложение. Однако в этом случае потребуются дополнительные инструменты.

На удаленном компьютере нужно будет запустить утилиту msvsmon.exe (Remote Debugging Monitor). Ее можно установить на жесткий диск или использовать для запуска разделяемую папку в сети. По умолчанию она запускается как обычное Windows-приложение, однако для использования в ASP.NET ее нужно будет запустить как службу (соответствующим образом сконфигурировать).

Вопросы удаленной отладки выходят за рамки нашего учебного курса, мы только обозначили такую возможность. При необходимости ее использования слушателям предлагается изучить соответствующую документацию.

Раздел 2. Трассировка в ASP.NET

Трассировка — это получение уведомляющих сообщений во время исполнения приложения. С их помощью можно не только обеспечить корректность работы, но и проанализировать производительность как программы в целом, так и отдельных ее участков.

При компиляции в любом режиме операторы трассировки остаются в программе (даже если они не нужны), поэтому по умолчанию при выполнении программ трассировка отключена, точнее, ее вывод игнорируется. Если этот вывод все же нужен, его следует разрешить явным образом.

Основным классом для использования данного механизма в Microsoft Visual Studio является класс **TraceContext**. Нам предлагаются средства, позволяющие:

- включить режим вывода диагностической информации о запросе на уровне страницы или на уровне всего приложения;
- самостоятельно (в коде) выводить любую информацию;
- отследить процесс исполнения кода для вашей страницы или приложения.

Класс TraceContext

Класс **TraceContext** позволяет выводить сообщения трассировки прямо на страницу или записывать их в память.

Соответствующий объект доступен через свойство Trace класса **Page**, так что его экземпляр создавать не требуется. Объект имеет тип **System.Web.HttpContext**, поэтому он содержит все необходимое для работы с протоколом HTTP. Наследников его создавать нельзя, так как он объявлен с модификатором sealed.

Класс предлагает для вывода три варианта метода Write и три идентичных варианта метода Warn. Разница между ними только в том, что последний выводит сообщения красным цветом. Формат перегрузок метода Write:

```
Trace.Write("Trace Message");
Trace.Write("Category 1", "Trace Message");
Trace.Write("Category 2", "Trace Message",
 new Exception("En Exception was thrown, because..."));
```

Последний вариант, как мы видим, позволяет дополнительно выбросить исключение. Категории, как и ранее, предназначены для сортировки сообщений.

В директиве Page через атрибут TraceMode можно задать способ сортировки сообщений: SortByTime или SortByCategory.

```
<%@ Page ... Trace="True" TraceMode="SortByCategory" %>
```

Свойство IsEnabled позволяет определить в коде программы, разрешена или запрещена трассировка:

```
if (Trace.IsEnabled)
{
    Trace.Write("Tracing is enabled!");
}
```

Это свойство не является свойством только для чтения, так что при желании его значение в коде программы можно изменить.

Сообщения трассировки, кроме прочего, содержат информацию о времени, прошедшем с начала работы приложения и с момента вывода предыдущего сообщения. Время выводится с очень большой точностью, что позволяет в том числе получить информацию о продолжительности выполнения отдельных участков кода.

Трассировка Web-приложения

Включить режим трассировки на уровне страницы можно в директиве Page:

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

Включение этого режима на уровне всего приложения выполняется через файл Web.config:

```
<configuration>
    ...
    <system.web>
        <trace enabled="true" />
        ...
    </system.web>
</configuration>
```

Чтобы сообщения трассировки не появлялись, не требуется вносить изменения в код, достаточно просто ее отключить.

На уровне приложения вы можете включить трассировку для всех страниц, на этом же уровне вы можете указать, что сообщения следует сохранять в памяти. Впоследствии они могут быть просмотрены с помощью специального вьюера trace.axd. Он вызывается по завершении работы приложения в браузере через запрос http://Ваше_Приложение/trace.axd.

В файле Web.config в элементе trace вы можете использовать также следующие атрибуты:

- pageOutput = "true", если сообщения нужно выводить на страницу;
- pageOutput = "false", если сообщения нужно сохранять в памяти;
- localOnly = "true", если сообщения были видны только на локальном компьютере.

Отметим, что кроме ваших сообщений трассировки в выводе присутствует довольно много сообщений системных (более десятка категорий), поэтому для удобства фильтрации рекомендуется для своих сообщений задавать удобные и информативные названия категорий.

Лекция 8. Управление данными Web-приложений ASP.NET

В данной теме мы рассмотрим основы работы с данными Web-приложений ASP.NET на основе ADO.NET. Это технология доступа к источникам данных со стороны .NET-приложений любого типа. В качестве источника данных могут выступать самые различные СУБД, например: Microsoft SQL Server, файлы баз данных, XML-файлы и многие другие.

ADO.NET специально разработана для доступа к данным без использования постоянного подключения. Она обеспечивает разработчикам .NET-приложений простой и гибкий способ встроить в них доступ к данным и их обработку.

Раздел 1. Обзор ADO.NET

Ввиду важности вопросов доступа к данным со стороны Web-приложения нам необходимо разобраться с механизмами и средствами, которые для этой цели нам предлагает ADO.NET. Этому и посвящен данный раздел. В нем мы, в частности, рассмотрим два основных компонента ADO.NET: поставщики данных и класс DataSet.

Что такое ADO.NET?

ADO расшифровывается как ActiveX Data Objects. ADO.NET нельзя назвать новой версией более ранней технологии ADO. Это скорее новый способ доступа к данным, более простой для разработчика и более мощный. При этом не исключается ее совместное использование со старой технологией ADO, впрочем, нам это здесь не потребуется.

С точки зрения содержимого мы может рассматривать ADO.NET как набор классов, интерфейсов, структур и перечислений для подключения к источникам данных и работы с ними. В этот набор также следует включить соответствующие службы.

В отличие от ADO, которая базировалась на технологии COM и использовала для доступа к данным соединение через OLE DB, ADO.NET для работы не требует наличия постоянного соединения, что особенно удобно для Web-приложений. Для передачи данных между источником и Web-приложением используется формат XML.

Инструменты ADO.NET можно распределить на два слоя: слой, работающий без соединения с источником данных, и слой, отвечающий за соединение.

Первый слой образуют классы, предназначенные для хранения самих данных и их отношений. Ключевым классом здесь можно назвать DataSet — своеобразный кэш базы данных в памяти. Он содержит коллекцию таблиц (класс DataTable), которые состоят из строк и столбцов данных, первичного ключа, внешнего ключа, ограничений. DataSet также может содержать коллекцию отношений между таблицами.

Второй слой, работающий в режиме соединения, привязан к специфике источника данных (конкретная СУБД или файл XML). Этот слой и соответствующий программный компонент принято называть поставщиком данных. ASP.NET, в частности, обеспечивает возможность взаимодействия с множеством поставщиков. Мы в основном будем использовать Microsoft SQL Server седьмой и более свежих версий. Однако можно работать с любой СУБД, поддерживающей интерфейсы OLE DB и ODBC. Кроме того, существует возможность подключения поставщиков данных от сторонних разработчиков.

Объектная модель ADO.NET

Объектная модель ADO.NET обеспечивает инфраструктуру для доступа к данным из источников самой разной природы. Как было сказано выше, все эти средства можно распределить по двум слоям:

- слой, не требующий соединения, это **DataSet** и связанные с ним классы;
- слой для организации соединения и обмена данными с реальным источником, называемый поставщиком данных.

Первый слой не зависит от источника и представляет фактически кэш данных в оперативной памяти. Он может представлять данные из разных источников, в том числе и нереляционных (XML-данные). Но для его наполнения или внесения изменений в источник потребуется использовать поставщик данных. Перечислим основные типы, представленные на первом слое.

- **Constraint**. Это ограничения, относящиеся к одному или более объектов **DataColumn**. Ограничение — это правило для поддержки целостности данных в таблице. **Constraint** — абстрактный класс, его наследниками являются классы **UniqueConstraint** и **ForeignKeyConstraint**.
- **DataColumn**. Класс представляет схему отдельного столбца в **DataTable**, набор таких объектов фактически определяет структуру всей таблицы.
- **DataRelation**. Представляет отношение типа «родитель – ребенок» между двумя таблицами. Отношения обеспечивают основу для навигации между связанными таблицами.
- **DataRow**. Представляет строку данных в объекте **DataTable**.
- **DataSet**. Главный контейнер для представления данных в памяти. Содержит внутри себя коллекции объектов классов **DataTable** и **DataRelation**. Класс **DataSet** является сериализуемым.
- **DataTable**. Представляет одну таблицу в **DataSet**. Структура данных представляется одним или более объектов **DataColumn**, а сами данные представлены в виде объектов **DataRow**. Это тоже сериализуемый класс.
- **DataTableReader**. Позволяет получить содержимое одного или более объектов **DataTable** в виде одного или более множеств «только для чтения», при этом навигация может осуществляться только вперед.
- **DataView**. Класс позволяет организовать заказное представление **DataTable**. Может использоваться для сортировки, фильтрации, поиска, редактирования, а также навигации.

Перечислим теперь типы, относящиеся ко второму слою (использующему соединение). Используемые ниже имена не являются реальными именами классов, поскольку реальные имена привязаны к конкретному источнику. Например, вместо «типа» **Connection** следует использовать класс **SqlConnection**, **OdbcConnection**, **OleDbConnection** или другой, в зависимости от используемого поставщика данных.

- **Command**. Класс обеспечивает доступ к данным: получение, модификацию данных, запуск хранимых процедур и т. п.
- **Connection**. Класс обеспечивает соединение с источником данных.
- **DataAdapter**. Это связующее звено между **DataSet** и источником данных. Он использует объекты **Command** для выполнения SQL-запросов, причем данные можно и читать, и писать.
- **DataReader**. Обеспечивает быстрый однонаправленный поток для чтения данных.

Обзор ADO.NET Entity Framework

Entity Framework — это множество технологий в ADO.NET, предназначенных для работы с данными в рамках платформы данных Microsoft data platform. Давайте разберемся, для решения каких проблем она предназначена.

Разработчики вынуждены работать с очень разными объектами и понятиями: сущностями, отношениями, они должны реализовывать бизнес-логику и иметь дело с механизмами СУБД для получения и загрузки данных (а каждая СУБД имеет свой протокол взаимодействия).

Entity Framework позволяет работать с данными на уровне «объекты и свойства», например заказчики и их адреса, не задумываясь при этом, в каких таблицах эти данные хранятся в конкретной базе данных. Можно сказать, что это такой уровень абстракции и соответствующий компонент .NET Framework.

Обычно при проектировании базы данных выделяют три модели: концептуальную, логическую и физическую:

- *концептуальная модель* определяет сущности и отношения моделируемой системы;
- *логическая модель* представляет их в виде таблиц с ограничениями внешних ключей;
- *физическая модель* имеет дело с возможностями и механизмами конкретной СУБД: декомпозицией, индексированием и т. п.

Настройкой физической модели администраторы могут улучшить производительность. Впрочем, здесь могут приложить

руку и разработчики: сформулировать грамотные SQL-запросы, создать хранимые процедуры.

Концептуальная модель строится на ранних стадиях проектирования и зачастую впоследствии забывается. А многие разработчики и вовсе этот этап пропускают и сразу проектируют таблицы, столбцы и ключи реляционной базы данных.

Entity Framework фактически возрождает концептуальные модели и освобождает нас от трудоемкого кодирования отношения зависимостей. Отображение концептуальной модели на уровень хранения данных осуществляется соответствующей оболочкой, которая построена по спецификации EDM (Entity Data Model).

Модель хранения, а значит, и отображение на нее впоследствии могут измениться, но это не потребует внесения изменений в концептуальную модель, классы для представления данных и в код приложения. Одна и та же концептуальная модель может быть использована при работе с базами данных разных поставщиков.

В EDM используются следующие типы файлов:

- файл описания концептуальной модели с расширением .csdl (conceptual schema definition language);
- файл описания модели хранения с расширением .ssdl (store schema definition language);
- файл описания модели отображения концептуальной модели на модель хранения с расширением .msl (mapping specification language).

При программировании объектно-ориентированных приложений нередко структура базы данных напрямую переносится в классы. С точки зрения объектно-ориентированного программирования это отнюдь не всегда хорошо, поскольку отношения между объектами и отношения между таблицами — это разные вещи.

Например, в классе **Order** есть свойство со ссылкой на экземпляр класса **Customer**. Таблица **Order** содержит внешний ключ (столбец или несколько столбцов), который соответствует первичному ключу в таблице **Customer**. В классе **Customer** разумно было бы иметь свойство **Orders** — коллекцию из объектов класса **Order**, но в таблице **Customer** такого столбца нет.

Entity Framework отображает реляционные таблицы, столбцы, ограничения внешних ключей логической модели в виде сущностей и отношений модели концептуальной.

Соответствующие инструменты EDM генерируют расширяемые классы на основе концептуальной модели. Они снабжаются модификатором partial, а значит, мы может добавлять к их объявлению что угодно. В качестве базовых используются классы, поддерживающие механизм «Object Services for materializing entities as objects», что позволяет нам работать с сущностями и отношениями, как с объектами.

Entity Framework позволяет приложениям иметь доступ к данным (и изменять их), которые представлены как сущности и отношения в концептуальной модели. Служба Object Services использует EDM, чтобы транслировать эти запросы в запросы, специфичные для конкретного источника данных, а их результаты опять материализуются в объекты.

Специальный контрол EntityDataSource осуществляет привязку к данным по спецификации EDM, представляющей данные как множество сущностей и отношений. Такая технология называется объектно-реляционным отображением — ORM (Object-relational mapping). В процессе разработки контрол EntityDataSource настраивается аналогично другим контролам доступа к данным. Он управляет операциями чтения, создания, замены и удаления в источнике данных или контролах из категории data-bound. Он может работать с редактируемыми сетками (grid), с формами, на которых пользователь задал сортировку и фильтрацию, со страницами master-detail, обеспечивая навигацию, и т. п.

Инструменты EDM разработаны с целью помочь разработчику создавать приложения Entity Framework и допускают несколько сценариев.

- Вы можете создать концептуальную модель по существующей базе данных, потом ее визуализировать и отредактировать.
- Вы можете сначала визуально построить концептуальную модель, а затем сгенерировать по ней базу данных.

В любом случае вы можете автоматически изменять свою модель, когда изменяется база и автоматически генерировать код своего приложения.

Кроме поддержки периода выполнения для Entity Framework .NET Framework, включает в себя 4 генератора EDM. Они реализованы в виде утилиты командной строки, которая может подключиться к источнику данных и сгенерировать концептуальную модель (.csdl), модель хранения (.ssdl) и их отображение (.msl). EDM-генератор может проверить существующую модель и сгенерировать код, скажем, на C#, который будет содержать классы для нашей концептуальной модели (по файлу .csdl).

Раздел 2. Подключение к базе данных

Проще всего можно установить соединения с базой данных, используя интегрированную среду разработки Microsoft Visual Studio, однако это можно сделать и программным путем. Для доступа к данным мы будем использовать объекты **DataAdapter** и **DataReader**.

Создание соединения

Как только что было сказано, самый простой способ для создания подключения к базе данных предоставляет среда Visual Studio. Для этого потребуется выполнить следующие шаги.

1. В окне **Server Explorer** выполнить щелчок правой кнопкой мыши на пункте **Data Connections** и выбрать вариант **Add Connection**.
2. В появившемся диалоге **Add Connection** в окне **Data source** выбрать подходящий вариант поставщика данных — это определит одновременно тип источника.
3. В появляющихся далее диалоговых окнах выбрать имя сервера базы данных, имя самой базы, способ проведения аутентификации и нажать кнопку **OK**.

После этого в окне **Server Explorer** отображается добавленное подключение. Оно позволяет просматривать структуру базы, данные ее таблиц, перетаскивать таблицы на формы и многое другое.

Альтернативный способ подключения — выполнить его программным путем. Рассмотрим необходимые для этого действия в предположении, что мы используем поставщик данных Microsoft SQL Server 7.0 или более новый, от этого выбора зависит префикс в именах типов.

Сначала подключим необходимое пространство имен:

```
using System.Data.SqlClient;
```

Далее создаем объект **Connection** подходящего типа:

```
SqlConnection orderConnection = new SqlConnection();
```

Затем формируем строку соединения (имя сервера и базы данных здесь условные):

```
orderConnection.ConnectionString =
                "Data Source=DatabaseServer; "+
                "Initial Catalog=OrderDatabase; "+
                "Integrated Security=True";
```

Самый простой способ сформировать правильную строку коннекции в учебном классе — подключиться к базе с использованием окна **Server Explorer**, а затем скопировать строку из созданного соединения.

До того, как соединение будет использоваться объектом **DataAdapter** или **DataReader**, его следует открыть:

```
orderConnection.Open();
```

Отметим, что соединение с базой данных относится к категории неуправляемых ресурсов, а значит, по завершении использования его следует явным образом закрыть:

```
// Закрыть соединение
orderConnection.Close();
// Освободить ресурсы
orderConnection.Dispose();
```

Для автоматического освобождения ресурсов удобно пользоваться оператором using:

```
using (SqlConnection orderConnection = new
                                       SqlConnection())
{
    orderConnection.ConnectionString =
                "Data Source=DatabaseServer; "+
                "Initial Catalog=OrderDatabase; "+
                "Integrated Security=True";
    orderConnection.Open();
    ...
    orderConnection.Close();
}
```

Вообще-то в этом случае даже нет необходимости вызывать метод Close, поскольку при выходе из блока автоматически вызывается метод Dispose, а он уже закроет все сам.

Организация передачи данных

Для этой цели могут использоваться объекты **DataAdapter** и **DataReader**. Последний в основном применяется в случаях, когда данные нужно просто получить из базы, не выполняя никаких манипуляций. Он работает быстро и потребляет мало ресурсов, однако очень ограничен в своих возможностях.

Программное создание объекта DataAdapter

Объект **DataAdapter** использует объект **Connection** для подключения к базе данных и набор объектов **Command** для выполнения действий по управлению данными. Всего этих объектов четыре, они доступны через свойства **SelectCommand**, **UpdateCommand**, **InsertCommand** и **DeleteCommand**. Их названия говорят сами за себя, обычно они основаны на использовании соответствующих операторов языка SQL.

Свойство **SelectCommand** обязательно должно быть установлено, остальные — по потребности. При попытке выполнить манипуляцию, за которую отвечает неустановленное свойство, будет выброшено исключение. Свойства **Command** не обязательно выполняют простой оператор SQL, их можно использовать также для запуска хранимых процедур.

Объект **DataAdapter** можно создать программно. Рассмотрим пример в предположении, что мы используем поставщик данных Microsoft SQL Server 7.0 или более новый. После подключения необходимого пространства имен создаем экземпляр **DataAdapter**:

```
// Создание экземпляра DataAdapter
SqlDataAdapter ordersDataAdapter = new
                                        SqlDataAdapter();
```

Далее можно создать объекты **Command** и инициализировать соответствующие свойства адаптера:

```
// Объявление и инициализация объектов Command
SqlCommand selectCommand =
    new SqlCommand("SELECT * FROM Orders",
                   orderConnection);
```

```
SqlCommand deleteCommand =
    new SqlCommand("DELETE FROM Orders WHERE ID=@ID",
                   orderConnection);
SqlCommand insertCommand =
    new SqlCommand("INSERT INTO Orders (CustomerID, " +
        "InvoiceDate, CreatedDate, CreatedBy) " +
        "VALUES(@CustomerID, @InvoiceDate, " +
        "@CreatedDate, @CreatedBy)",
        orderConnection);
SqlCommand updateCommand = new SqlCommand(
        "UPDATE Orders SET CustomerID=@CustomerID, " +
        "InvoiceDate=@InvoiceDate, " +
        "ModifiedDate=@ModifiedDate, " +
        "ModifiedBy=@ModifiedBy WHERE ID=@ID", " +
        "orderConnection);
// Инициализируем свойства ordersDataAdapter
ordersDataAdapter.SelectCommand = selectCommand;
ordersDataAdapter.DeleteCommand = deleteCommand;
ordersDataAdapter.InsertCommand = insertCommand;
ordersDataAdapter.UpdateCommand = updateCommand;
```

В этом примере объект **selectCommand** использует оператор SQL SELECT для получения всех строк в таблице Orders.

Объект **deleteCommand** удаляет все строки, которые содержат заданное значение @ID в столбце ID. @ID — это именованный параметр, он получит свое значение позднее. В таблице Orders значение ID уникально, так что будет удалена только одна строка.

Объект **updateCommand** также использует параметр @ID, он записывает значения в различные столбцы (CustomerID = @CustomerID) и т. п. Аналогичным образом поступает и объект **insertCommand**.

Рассмотрим пример кода для передачи параметров.

```
// Объявление и создание объектов Parameter
SqlParameter deleteIDParameter =
    new SqlParameter("@ID", SqlDbType.UniqueIdentifier,
                     0, "ID");
SqlParameter updateIDParameter =
    new SqlParameter("@ID", SqlDbType.UniqueIdentifier,
                     0, "ID");
```

```
SqlParameter updateInvoiceDateParameter =
    new SqlParameter("@InvoiceDate",
                     SqlDbType.SmallDateTime, 0,
                     "InvoiceDate");
SqlParameter insertInvoiceDateParameter =
    new SqlParameter("@InvoiceDate",
                     SqlDbType.SmallDateTime, 0,
                     "InvoiceDate");
...
// Добавление их в коллекцию объекта Command
deleteCommand.Parameters.Add(deleteIDParameter);
updateCommand.Parameters.Add(updateIDParameter);
insertCommand.Parameters.Add(insertInvoiceDateParameter
);
```

Здесь можно заметить, что, казалось бы, одинаковые параметры дублируются. Это действительно нужно, поскольку они добавляются к разным объектам **Command**.

Замечание. Для работы с одной таблицей, чтобы не создавать вручную все объекты **Command**, можно создать только **SelectCommand** (он самый простой), а затем сгенерировать остальные три, воспользовавшись классом **CommandBuilder**.

Программное создание объекта DataReader

DataReader представляет собой однонаправленный поток для чтения, он работает очень быстро. Соединение с базой данных требуется открывать и закрывать вручную.

Рассмотрим пример в предположении, что мы используем поставщик данных Microsoft SQL Server 7.0 или новее. Объект **DataReader** создается через вызов метода ExecuteReader объекта **Command**.

Здесь мы, используя созданный ранее объект **selectCommand**, получаем все строки из таблицы Orders:

```
SqlDataReader ordersDataReader =
                     selectCommand.ExecuteReader();
// Есть очередная строка?
while (ordersDataReader.Read())
{
    ...
}
// Вызов Close по окончании
ordersDataReader.Close();
```

К моменту вызова ExecuteReader соединение с источником данных, разумеется, должно быть открыто. Если это не так, будет выброшено исключение.

Для чтения данных используется метод Read. Он возвращает true, если текущая позиция для чтения установлена на правильной строке. Если он возвратил false, значит, все уже прочитано.

Отметим еще раз, что при использовании **DataReader** мы можем двигаться только вперед. При этом никто не запрещает нам закрыть поток, а потом открыть заново для повторного чтения.

Раздел 3. Управление данными

После установления соединения с источником данных и их получения надо как-то показать данные пользователю и предоставить ему возможность работы с ними. Для этой цели в ADO.NET есть средства и на «соединенном», и на «рассоединенном» слое.

Наибольшие возможности предоставляет объект **DataSet** — своего рода кэш данных в оперативной памяти. Если с данными нужно работать долго и выполнять много операций, это лучший выбор.

Но бывают и более простые задачи: показать некоторый набор данных пользователю, почитать из источника правильный пароль и т. п. Для таких задач вполне годится и **DataReader**.

Получение простых данных

Рассмотрим два простых варианта:

- если вам требуется получить *единственное* значение (скажем, пароль или количество строк в таблице), можно использовать метод ExecuteScalar класса **SqlCommand**;
- если нужно просто получить данные (в цикле) для показа пользователю, то лучший выбор — класс **DataReader**.

Использование класса **DataReader** для получения данных выглядит примерно так.

- Используем метод ExecuteReader объекта **Command**. Он посылает источнику данных команду SELECT, а результат возвращает через объект **DataReader**.

- Далее доступ собственно к данным осуществляется через вызов метода Read объекта **DataReader**. При этом если SELECT возвращает несколько результирующих множеств, для получения следующего можно использовать метод NextResult.

Рассмотрим пример кода:

```
// Объявляем и создаем соединение
SqlConnection orderConnection = new SqlConnection();
// Инициализируем соединение
orderConnection.ConnectionString = "...";
// Объявляем и создаем объект Command
SqlCommand selectCommand =
    new SqlCommand("SELECT * FROM Orders",
    orderConnection);
// Открываем соединение
orderConnection.Open();
// Объявляем, создаем и инициализируем DataReader
SqlDataReader ordersDataReader =
    selectCommand.ExecuteReader();
// Есть ли хоть одна строка?
if (!ordersDataReader.HasRows)
{
    ...
}
else
{
    // Есть ли еще строки?
    while (ordersDataReader.Read())
    {
    ...
    }
}
// Закрываем DataReader
ordersDataReader.Close();
// Закрываем соединение
orderConnection.Close();
```

Свойство HasRows объекта **DataReader** позволяет нам узнать, была ли возвращена хотя бы одна строка, прежде чем мы попытаемся их читать в цикле. После того как метод Read вернет

false, попытка доступа к данным в объекте **DataReader** приведет к выбрасыванию исключения.

Если нужно прочитать единственное значение, лучше воспользоваться методом ExecuteScalar. Рассмотрим такой запрос:

```
SELECT ID FROM Orders WHERE CreatedBy='Weber'
```

Если существует заказчик по имени Weber, этот запрос возвратит идентификатор его заказа. Если таких записей несколько, мы получим только одно, первое попавшееся значение. Какое значение окажется «первым», зависит от того, каким образом отсортированы строки в таблице Orders.

Если в SELECT сделан запрос на получение нескольких полей, метод ExecuteScalar вернет только первый столбец.

Отметим также, что метод ExecuteScalar возвращает значение **System.Object**, а значит, его еще нужно привести к правильному типу.

Рассмотрим пример использования этого метода.

```
// Объявляем и создаем соединение
SqlConnection orderConnection = new SqlConnection();
// Инициализируем соединение
orderConnection.ConnectionString = "...";
// Объявляем и создаем объект Command
SqlCommand selectCommand =
    new SqlCommand(
       "SELECT ID FROM Orders WHERE CreatedBy='Weber'",
       orderConnection);
// Открываем соединение
orderConnection.Open();
// Получаем скалярное значение
string name = (string) selectCommand.ExecuteScalar();
// Закрываем соединение
orderConnection.Close();
```

Получение более сложных данных

Если вы собираетесь работать с более сложными данными — содержимым целых таблиц или с некоторым набором строк из таблиц, — то самое удобное — использовать классы **DataSet** и **DataTable**. Отметим дополнительные преимущества такого подхода:

- для обеспечения целостности данных в объекте **DataSet** можно использовать ограничения;
- **DataSet** можно создавать и использовать для хранения локальных данных, добавляя их непосредственно в коде приложения;
- **DataSet** позволяет легко импортировать XML-документы.

Использование DataSet

Рассмотрим примеры кода с использованием **DataSet**. Создание экземпляра выглядит очень привычно:

```
DataSet ordersDataSet = new DataSet();
```

Конструкторов класс **DataSet** имеет несколько, здесь использован самый простой вариант.

Далее его следует заполнить данными. Для этого нам потребуется объект класса **DataAdapter** и его метод Fill. Этот метод создает и заполняет таблицу внутри объекта **DataSet**. Пример:

```
ordersDataAdapter.Fill(ordersDataSet, "Orders");
```

Метод Fill запускает на выполнение SQL-оператор из свойства SelectCommand объекта **DataAdapter**. Имя таблицы (второй параметр) лучше задавать всегда, оно пригодится нам для доступа к данным. Для каждой таблицы, которую мы хотим считать, нам потребуется отдельный экземпляр типа **DataAdapter**.

Метод Fill неявным образом использует объект **DataReader** для получения имен столбцов и их типов при создании таблиц в **DataSet**. Данными она будет заполнена только при условии, что таковые уже есть в источнике, иначе будет получена только схема таблицы.

Доступ к таблицам

После заполнения **DataSet** содержит внутри себя объекты **DataTable**. Обращаться к ним можно по имени (что удобнее) или по номеру (в порядке создания), как показано в примере:

```
// Выражение для доступа к таблице Orders по имени
ordersDataSet.Tables["Orders"]
// Выражение для доступа к таблице по номеру
ordersDataSet.Tables[0]
```

Основными компонентами **DataTable** являются объекты классов **DataRow** и **DataColumn**, доступные через свойства Rows

и Columns (коллекции **DataRowCollection** и **DataColumnCollection** соответственно). Строки здесь определяют собственно данные, а столбцы задают структуру таблицы. Пример доступа к столбцам и вывода их имен:

```
foreach (DataColumn col in
                ordersDataSet.Tables["Orders"].Columns)
{
    Response.Write(col.ColumnName);
};
```

Разумеется, обе коллекции имеют свойство Count; пример использования не приводим, ввиду его очевидности.

Отметим, что использовать индекс для идентификации строки в таблице не слишком разумно. Их порядок не обязательно соответствует порядку в источнике, строки могут быть отсортированы различным образом или помечены для удаления.

Для поиска нужной строки можно, конечно, использовать цикл, но умнее будет использовать метод **Find**, который осуществляет поиск по первичному ключу:

```
// Получить схему из источника данных
ordersDataAdapter.FillSchema(ordersDataSet,
                          SchemaType.Source, "Orders");
// Получить требуемую строку
DataRow ordersDataRow =
       ordersDataSet.Tables["Orders"].Rows.Find(
              "a05c7f53-9c4e-de11-aa78-0003ffa70544");
```

Обратите здесь внимание на вызов метода FillSchema для получения схемы таблицы Orders объекта **DataTable**. Если вы планируете использовать в программе первичный ключ, такой вызов обязательно нужно сделать.

Манипуляции с данными

Для тех или иных преобразований данных, локально или на сервере, мы будем использовать объекты **DataAdapter** и **Command**. Для этого сначала нужно выполнить изменения в данных, загруженных в оперативную память, а затем сохранить измененные данные в конечном источнике.

Разберем несколько примеров обычных действий с данными в оперативной памяти. Для начала посмотрим, как можно добавить строку:

```
DataRow newOrderDataRow =
            ordersDataSet.Tables["Orders"].NewRow();
newOrderDataRow["ID"] = Guid.NewGuid();
newOrderDataRow["CustomerID"] = currentCustomer.ID;
newOrderDataRow["InvoiceDate"] = DateTime.Now;
newOrderDataRow["CreatedDate"] = DateTime.Now;
newOrderDataRow["CreatedBy"] = currentUser.Name;
```

Обратите внимание, что новая строка создается через метод класса **DataTable**, поскольку именно этот класс имеет достаточную информацию об устройстве таблиц. Теперь рассмотрим пример изменения строки:

```
// Найти строку для изменения
DataRow ordersDataRow =
       ordersDataSet.Tables["Orders"].Rows.Find(
              "a05c7f53-9c4e-de11-aa78-0003ffa70544");
// Изменить локально
orderDataRow["ModifiedDate"] = DateTime.Now;
orderDataRow["ModifiedBy"] = currentUser.Name;
```

После внесения необходимых изменений мы можем сохранить их в источнике данных:

```
ordersDataAdapter.Update(ordersDataSet, "Orders");
```

Контролы для управления данными

Для эффективного управления данными Visual Studio предоставляет нам ряд специальных контролов, некоторые из которых мы здесь рассмотрим.

К числу самых популярных контролов из этой категории можно отнести элемент управления **GridView**. Для его создания достаточно сразу после подключения к базе данных перетащить нужную таблицу из окна Server Explorer в окно формы, открытой в режиме Design view. Среда разработки создаст не только объект **GridView** для работы с этой таблицей, но и невидимый контрол **SqlDataSource**, с нею связанный. Идентификаторы этих контролов далее могут использоваться в коде программы для управления ими. Также среда разработки записывает соответствующую строку коннекции в файл Web.config:

```
<configuration>
    ...
    <appSettings/>
    <connectionStrings>
        <add name="OrderConnectionString1"
                                    connectionString=.../>
    </connectionStrings>
    ...
</configuration>
```

Контрол **SqlDataSource** используется в связке с так называемыми data-bound контролами, одним из них является **GridView**. В разметке страницы эта связь выглядит примерно так:

```
<asp:GridView ID="GridView1" runat="server"
          AutoGenerateColumns="False"
          DataKeyNames="ID"
          DataSourceID="SqlDataSource1"
          EmptyDataText=
              "There are no data records to display.">
```

Свойства контролов, разумеется, удобнее всего настраивать в окне Properties. Перечислим их основные свойства. Начнем со свойств контрола **SqlDataSource**:

- *ConnectionString* — строка коннекции;
- *ProviderName* — имя сборки поставщика данных.

Основные свойства контрола **GridView**:

- *AllowPaging* — разрешение постраничного вывода;
- *AllowSorting* — разрешение сортировки данных;
- *Columns* — задает множество столбцов для показа;
- *DataSourceID* — идентификатор источника данных;
- *PageSize* — количество строк для вывода на странице.

Кроме SqlDataSource, для подключения к источнику данных можно использовать и некоторые другие контролы, например **LinqDataSource**. Технология LINQ радикально упрощает взаимодействие между объектно-ориентированным кодом и реляционными данными в источнике.

Лекция 9. Использование LINQ для управления данными

Первый раздел данной темы посвящен краткому обзору основных моментов использования технологии LINQ, изучавшейся нами ранее.

Далее мы рассмотрим вариацию LINQ to XML. Хотя огромное количество данных информации хранится в реляционных базах, XML фактически стал стандартом для хранения, управления и передачи информации. В то же время XML и, скажем, SQL Server требуют разных методов для управления данными.

В то же время для запроса данных из любого источника мы можем использовать средства LINQ, причем и запросы, и получаемые результаты будут строго типизированными. Кроме того, LINQ облегчает жизнь разработчиков благодаря поддержке со стороны IntelliSense и проверке на наличие ошибок во время компиляции.

Мы разберем, как можно использовать LINQ для управления XML-данными и данными из реляционных источников в ASP.NET Web-приложениях с использованием Web-серверных контролов и программного кода.

Последний раздел посвящен основам использования вариаций LINQ to SQL и LINQ to Entities, а также их сравнению.

Раздел 1. Обзор LINQ

Мы исходим из того, что технология LINQ изучена ранее. В данном разделе мы лишь кратко вспомним основные моменты данной технологии. Более полную информацию по этому вопросу можно найти, например, в [5]. Поскольку все примеры далее ориентированы на использование SQL Server, мы будем везде упоминать ее в качестве источника реляционных данных, имея в виду, что почти все сказанное в полной мере относится и к прочим СУБД.

Способ доступа к данным SQL Server и данным XML очень сильно разнится. И до появления LINQ не было общего способа для доступа к данным из принципиально разных по своей природе источников. Также не было и единого прикладного интерфейса (API) для программного выполнения таких действий.

Использование LINQ позволяет нам все делать единым образом, причем синтаксические различия для кода, работающего с разными типами источников данных, минимальны. Далее мы:

- вспомним, как LINQ to XML позволяет создавать запросы к хранящимся в памяти XML-документам для получения коллекций элементов и атрибутов;
- вспомним, как использовать LINQ to SQL для создания SQL-запросов для выборки, фильтрации и группировки данных от поставщика SQL Server;
- узнаем, что собой представляет вариация LINQ to Entities и как ее использовать.

Что нам дает использование LINQ?

LINQ предлагает удобную программную модель для доступа к данным с использованием различных языков программирования в .NET Framework. Она является своего рода мостиком между объектами в объектно-ориентированных приложениях и данными в различных источниках.

Прежде запросы оформлялись в виде простой строки, причем для каждого типа источника это была своя строка, поддержка со стороны IntelliSense отсутствовала.

Сейчас, используя LINQ, мы можем формулировать запросы с использованием синтаксиса, совпадающего (или очень близкого) с синтаксисом своего объектно-ориентированного языка программирования. В нашем случае это, конечно, язык C#.

При этом синтаксис самих запросов будет единым для разных типов источников данных. Это и XML-данные, и данные от SQL Server, и прочие данные ADO.NET, и любые коллекции объектов в оперативной памяти, в том числе обобщенные, реализующие интерфейс IEnumerable.

Для поддержки LINQ Visual Studio предоставляет нам следующие инструменты:

- специальные редакторы кода с поддержкой IntelliSense и специфического формата LINQ;
- поддержку запросов LINQ со стороны отладчика.

Для того чтобы читателям было проще вспомнить, как выглядят запросы LINQ, приведем пример запроса для перечисления

всех файлов в подпапке Public корневой папки нашего Web-приложения:

```
var files = from file in System.IO.Directory.GetFiles(
                         Request.PhysicalApplicationPath +
                         "\\Public")
            select file;
```

Напомним также, что поддержка данной технологии осуществляется несколькими сборками, соответствующими различным вариациям LINQ:

- *LINQ to DataSet* — доступ к данным, хранящимся в объекте **DataSet**;
- *LINQ to Entities* — доступ к реляционным данным в соответствии с моделью EDM (Entity Data Model);
- *LINQ to Objects* – доступ к данным в памяти (массивы и коллекции);
- *LINQ to SQL* — доступ к реляционным данным с использованием специального типа контекста данных LINQ to SQL;
- *LINQ to XML* — доступ к данным XML-документа.

Операции запроса LINQ

Обычные операции представляют собой методы, которые образуют шаблон LINQ. Они обеспечивают такие возможности запроса, как фильтрация, агрегирование, сортировка. Можно выделить два множества стандартных операций запроса в LINQ:

- множество операций над объектами типа **IEnumerable<T>**;
- множество операций над объектами типа **IQueryable<T>**.

Соответствующие методы являются статическими членами классов **Enumerable** и **Queryable** соответственно. Они определены как методы расширения над типами, к которым относятся. Это означает, что они могут быть вызваны и как статические методы, и как экземплярные.

Время выполнения запроса может сильно разниться в зависимости от того, возвращает ли он единственное значение или целую последовательность. В первом случае мы результат получаем сразу. Если нужно получить последовательность, возвращается перечисляемый объект. Большинство операций

имеют дело с последовательностями, то есть с объектами, тип которых реализует интерфейс **IEnumerable<T>** или **IQueryable<T>**.

Следующий пример демонстрирует, как обычный запрос используется для получения последовательности значений. Здесь приведены два варианта одного и того же запроса: с использованием синтаксиса операций и с использованием методов расширения.

```
string sentence =
        "the quick brown fox jumps over the lazy dog";

// Разбиваем строку на отдельные слова
string[] words = sentence.Split(' ');

// Использование синтаксиса операций
var query = from word in words
            group word.ToUpper() by word.Length into gr
            orderby gr.Key
            select new { Length = gr.Key, Words = gr };

// Использование методов расширения
var query2 = words.
             GroupBy(w => w.Length, w => w.ToUpper()).
             Select(g =>
                    new { Length = g.Key, Words = g }).
             OrderBy(o => o.Length);

foreach (var obj in query)
{
    Response.Write(string.Format(
        "Words of length {0}:<br />", obj.Length));
    foreach (string word in obj.Words)
        Response.Write(word + "<br />");
}
```

Полученный результат в обоих вариантах выглядит так:

```
Words of length 3:
THE
FOX
THE
DOG
Words of length 4:
```

```
OVER
LAZY
Words of length 5:
QUICK
```

Что такое LINQ to XML?

XML сейчас очень широко используется как средство хранения и передачи структурированной информации. Поэтому будет полезно разобраться, как LINQ работает с данными в этом формате и что собой представляет вариация LINQ to XML. Попробуем дать ее краткую характеристику.

LINQ to XML обеспечивает программный интерфейс поддержки XML, эксплуатирующий возможности .NET Framework. Он сравним с интерфейсом DOM (document object model). Сходство LINQ to XML и DOM заключается в том, что оба работают с XML-документами, хранящимися в памяти. Различие — в том, что LINQ to XML использует новую, более удобную программную модель, реализующую все преимущества языков .NET. Их конструкции позволяют получать из XML-документа целые коллекции элементов и атрибутов.

LINQ to XML также облегчает управление XML-данными, так как его запросы похожи на операторы запросов SQL. По функциональности LINQ to XML аналогичен языку XPath (XML Path Language), предназначенному для навигации по XML-документу и выбору отдельных узлов. Его также можно использовать для вычисления некоторых значений (числовых или булевских) по содержимому XML-документа.

В качестве примера запроса LINQ to XML рассмотрим получение элемента TLD (Internet top-level domain) для каждого узла Country в корневом элементе Countries:

```
var internetTLDs =
      from country in countries.Elements("Country")
      select (string)country.Element("InternetTLD");
```

Сам XML-документ может быть создан в памяти, например, так:

```
XElement countries = new XElement("Countries",
    new XElement("Country",
        new XElement("ID",
            "62750c5e-092c-de11-8c6e-0003ff4ed632"),
```

```
            new XElement("Name", "Uganda"),
            new XElement("PhoneNoFormat", ""),
            new XElement("DialingCountryCode", "256"),
            new XElement("InternationalDialingCode",
                         "000"),
            new XElement("InternetTLD", "UG")
        ),
        new XElement("Country",
            new XElement("ID",
                "602cf8a8-b62f-de11-a6b1-0003ffa70544"),
            new XElement("Name", "Denmark"),
            new XElement("PhoneNoFormat", "## ## ## ##"),
            new XElement("DialingCountryCode", "45"),
            new XElement("InternationalDialingCode", "00"),
            new XElement("InternetTLD", "DK")
        ),
        new XElement("Country",
            new XElement("ID",
                "63750c5e-092c-de11-8c6e-0003ff4ed632"),
            new XElement("Name", "Great Britain"),
            new XElement("PhoneNoFormat", ""),
            new XElement("DialingCountryCode", "44"),
            new XElement("InternationalDialingCode", "00"),
            new XElement("InternetTLD", "UK")
        )
);
```

Что такое LINQ to SQL?

Конечно, чаще всего информация хранится в реляционных базах данных под управлением той или иной СУБД, например SQL Server. При этом существует большая разница в представлении данных на языках программирования и реляционных базах данных.

LINQ to SQL позволяет нам в программном коде работать с реляционными данными, как с объектами. При этом модель данных традиционной базы данных отображается на объектную модель, представленную синтаксисом используемого языка программирования.

Во время выполнения программы LINQ to SQL транслирует запросы LINQ объектной модели в SQL-запросы и отправляет

базе данных для исполнения. После получения результаты конвертируются обратно в объекты.

В среде разработки Visual Studio вы можете использовать O/R Designer (Object Relational Designer) для создания контекста данных, который и занимается этим объектно-реляционным отображением в обе стороны. Создать контекст данных можно или через интерфейс пользователя, или (что проще) путем перетаскивания таблиц и объектов из окна Server Explorer на панель O/R Designer.

После добавления таблиц на поверхность O/R Designer вы можете видеть и устанавливать свойства классов, в которые преобразовались таблицы.

При компиляции вашего проекта специальная утилита (SqlMetal.exe) автоматически генерирует код классов, которые вы можете использовать для доступа к данным как к строго типизированным объектам.

Приведем кусочек кода, демонстрирующего такой доступ к данным. Мы создаем запрос на получение всех заказчиков из таблицы Customers, используя созданное ее отображение на объект **DataContext**.

```
// Объявление и создание экземпляра DataContext
CustomerManagementDataContext cmDataContext =
                    new CustomerManagementDataContext();
// Запрос на получение заказчиков из таблицы Customers
var query = from customers in cmDataContext.Customers
            select customers;
// Выводим имена найденных заказчиков
foreach (Customer cust in query)
                  Response.Write(cust.Name + "<br />");
```

Класс **CustomerManagementDataContext** (наследник класса **DataContext**) был создан утилитой SqlMetal.exe, мы можем с ним работать как с подключением ADO.NET.

После завершения использования экземпляра этого класса его следует освободить, вызвав метод **Dispose**. Если же этот объект вам требуется в приложении везде, его можно объявить глобально, например, в файле Global.asax.

Что такое LINQ to Entities?

Entity Framework тоже позволяет разработчикам обращаться с данными, как с объектами и свойствами конкретной области применения и соответствующей терминологией, например заказчиками и их адресами. При этом нас не заботит устройство таблиц и столбцов источника данных, где они хранятся.

LINQ to Entities позволяет писать запросы к данным на привычном языке программирования, а не в синтаксисе концептуальной модели Entity Framework, где они представлены в виде так называемых "command tree" запросов. LINQ to Entities транслирует наши запросы с C# в запросы "command tree" и переводит нам ответы.

Через специальную инфраструктуру Object Services ADO.NET обеспечивает представление данных, обычное для какой-либо концептуальной модели (включая и реляционные) в виде объектов в .NET-окружении.

Класс **ObjectContext** является базовым в модели EDM (Entity Data Model). Разработчик может создать для запросов данных экземпляр обобщенного класса **ObjectQuery** с использованием **ObjectContext** или его наследника. **ObjectQuery** представляет запрос, который возвращает экземпляр или коллекцию типизированных сущностей. При использовании изменять их нельзя, так как они размещены в контексте объекта **ObjectContext**.

Приведем пример использования LINQ to Entities для получения названий товаров из базы данных AdventureWorks:

```
using (AdventureWorksEntities AWEntities =
                          new AdventureWorksEntities())
{
    var productNames =
        AWEntities.Products.Select(p => p.Name);

    Response.Write("Product Names:" + "<br />");
    foreach (String productName in productNames)
    {
        Response.Write(productName + "<br />");
    }
}
```

Раздел 2. Управление данными с использованием LINQ to XML

XML сейчас широко используется для форматирования информации при записи или чтении из баз данных. Однако XML — это способ хранения данных, не самый простой в управлении, будь то модель DOM, другие объектные модели, инструменты или утилиты для работы с этим форматом данных.

LINQ to XML радикально все изменяет. Его нотация узлов, элементов и атрибутов позволяет намного проще запрашивать и обновлять данные. LINQ to XML — это программный интерфейс, который поддерживает выражения LINQ для чтения, записи, обработки XML-данных. Можно создавать запросы к хранящемуся в памяти XML-документу и получать коллекции элементов и атрибутов. Их можно модифицировать, сохранять в файл или сериализовать для передачи в сети.

Запрос XML-данных с использованием LINQ to XML

Отметим для начала, что природа хранилища XML-данных, как и любых других, может быть разной. Их можно разместить в оперативной памяти в виде строки или специальных объектов, они могут располагаться в каком-то внешнем хранилище, например в файле. Посмотрим, какие варианты хранения и доступа к таким данным предлагает нам .NET Framework. Их три:

- класс **System.Xml.XmlDocument**, реализующий модели DOM консорциума W3C (World Wide Web Consortium);
- класс **System.Xml.XPathDocument**, обеспечивающий быстрое, предназначенное только для чтения представление в памяти XML-документа с использованием модели XPath;
- вариацию LINQ to XML.

Первые два варианта используют нотацию документа как контейнера XML. В отличие от LINQ to XML они хорошо работают на уровне отдельных элементов с использованием класса **XElement**.

Для чтения можно использовать класс **System.Xml.XmlReader**. Он работает быстро, без кэширования, обеспечивая навигацию только вперед. Для выбора заданных отдельных элементов его использовать нельзя, разве что в цикле перебирать все элементы

в поисках нужного. Это можно, правда, сделать через запросы к содержимому объекта **XPathDocument** с применением языка XPath, однако использовать LINQ все же проще. Производительность LINQ to XML выше, чем у классов DOM, но, конечно, ниже, чем при использовании **XmlReader**.

При работе с XML нужно сначала определиться, где мы будем хранить данные: в строке, в файле или где-то еще. Рассмотрим следующий пример XML-данных:

```
<Countries>
    <Country>
        <ID>602cf8a8-b62f-de11-a6b1-0003ffa70544</ID>
        <Name>Denmark</Name>
        <PhoneNoFormat>## ## ## ##</PhoneNoFormat>
        <DialingCountryCode>45</DialingCountryCode>
        <InternationalDialingCode>00
        </InternationalDialingCode>
        <InternetTLD>DK</InternetTLD>
    </Country>
    <Country>
        <ID>62750c5e-092c-de11-8c6e-0003ff4ed632</ID>
        <Name>Uganda</Name>
        <PhoneNoFormat></PhoneNoFormat>
        <DialingCountryCode>256</DialingCountryCode>
        <InternationalDialingCode>000
        </InternationalDialingCode>
        <InternetTLD>UG</InternetTLD>
    </Country>
    <Country>
        <ID>63750c5e-092c-de11-8c6e-0003ff4ed632</ID>
        <Name>Great Britain</Name>
        <PhoneNoFormat></PhoneNoFormat>
        <DialingCountryCode>44</DialingCountryCode>
        <InternationalDialingCode>00
        </InternationalDialingCode>
        <InternetTLD>UK</InternetTLD>
    </Country>
</Countries>
```

Эти данные можно запрашивать независимо от способа их хранения. До появления LINQ to XML чтение из файла обычно осуществлялось с использованием класса, производного от абстрактного класса **System.Xml.XmlReader**. Если это возможно, так

следует поступать и сейчас, напрямую или опосредованно. Например, если приведенные данные содержатся в файле Countries.xml, можно для доступа использовать такой код:

```
XElement countries =
        XElement.Load(Server.MapPath("Countries.xml"));
```

Если же эта информация хранится в памяти в виде строки, вы можете использовать метод Parse класса **XElement**:

```
string xmlString =
    "<Countries><Country>...</Country></Countries>";
XElement countries = XElement.Parse(xmlString);
```

В обоих случаях результатом будет внешний элемент класса **XElement**, который содержит вложенные элементы **XElement** с нашими XML-данными.

Теперь из полученного объекта **countries** можно запрашивать информацию, например, так:

```
// Получить все элементы, содержащие PhoneNoFormat
var countriesWithPhNoFormat =
    (from c in countries.Elements()
     where c.Element("PhoneNoFormat") != null &&
         (string)c.Element("PhoneNoFormat").Value != ""
     select c);
```

В построенном запросе мы используем метод Elements объекта **countries** для получения коллекции потомков и выбираем из них те элементы, которые имеют вложенный элемент с именем PhoneNoFormat, имеющий непустое значение. В запросе используется свойство Element родительского объекта типа XElement для указания потомка.

Если бы нам требовалось получить более одного вложенного элемента, можно было бы воспользоваться методом Elements или Descendants. Descendants, в отличие от Elements, позволяет получить коллекцию всех потомков, а не только относящихся к одному поколению. Рассмотрим пример использования метода Descendants. Здесь для получения первого элемента использован знакомый нам метод расширения First:

```
var countriesWithPhNoFormat =
    from c in countries.Elements()
     where c.Descendants("PhoneNoFormat") != null &&
      (string)c.Descendants("PhoneNoFormat").
                                    First().Value != ""
     select c;
```

Данные XML не обязательно используют только вложенные элементы для задания иерархии, они могут часть информации хранить в виде атрибутов, например, так:

```
<Countries>
    <Country ID="38f5ad4b-092c-de11-8c6e-0003ff4ed632"
        Name="Denmark" PhoneNoFormat="#### ## ##"
        DialingCountryCode="45"
        InternationalDialingCode="00 "
        InternetTLD="DK"
    />
    <Country ID="62750c5e-092c-de11-8c6e-0003ff4ed632"
        Name="Uganda" PhoneNoFormat=""
        DialingCountryCode="256"
        InternationalDialingCode="000 "
        InternetTLD="UG"
    />
    <Country ID="63750c5e-092c-de11-8c6e-0003ff4ed632"
        Name="Great Britain"
        PhoneNoFormat="" DialingCountryCode="44"
        InternationalDialingCode="00 "
        InternetTLD="UK"
    />
</Countries>
```

В этом случае сделать аналогичную выборку можно так:

```
var countriesWithPhNoFormat =
    from c in countries.Elements()
    where c.Attribute("PhoneNoFormat") != null &&
      (string) c.Attribute("PhoneNoFormat").Value != ""
    select c;
```

Метод Attribute возвращает объект типа **XAttribute**, а метод Attributes позволяет получить более одного атрибута аналогично методу Elements.

Работа с XML-данными с использованием LINQ to XML

Если ваш XML-контент загружен в какое-либо долговременное хранилище, например в файл, мало уметь просто извлекать информацию, нужно еще иметь возможность изменять его содержимое.

Если вы читаете XML-документ с использованием класса **XmlReader**, то аналогичным образом вы можете задействовать

для его записи класс **XmlWriter**, однако для чтения и записи легче и естественней использовать соответствующие методы класса **XElement**.

Следующий пример показывает, как можно создать запрос на выборку стран, для которых задан формат телефонных номеров, а затем сохранить результат в файле CountriesWithPhNoFormat.xml:

```
// Получить все элементы, содержащие PhoneNoFormat
XElement countriesWithPhNoFormat =
   new XElement("Countries",
      from c in countries.Elements()
      where c.Element("PhoneNoFormat") != null &&
        (string) c.Element("PhoneNoFormat").Value != ""
      select new XElement("Country",
        new XElement(c.Element("ID")),
        new XElement(c.Element("Name")),
        new XElement(c.Element("PhoneNoFormat")),
        new XElement(c.Element("DialingCountryCode")),
        new XElement(c.Element(
                          "InternationalDialingCode")),
        new XElement(c.Element("InternetTLD"))
      )
   );
// Записать в файл
countriesWithPhNoFormat.Save(
        Server.MapPath("CountriesWithPhNoFormat.xml"));
```

Классы **XElement** и **XDocument**, конечно, имеют методы Load и Save, однако объект типа **XElement** еще надо было создать. Наш запрос LINQ возвращает не **XElement**, как хотелось бы, а объект, реализующий интерфейс **IEnumerable<XElement>**. Поэтому нам приходится создавать необходимый внешний объект и все вложенные в него объекты с помощью оператора new. Конструктор класса **XElement** в качестве первого аргумента принимает везде имя создаваемого узла.

Рассмотрим теперь примеры различных манипуляций с данными XML. Пусть изначально они выглядят так:

```
<Countries>
    <Country>
        <ID>602cf8a8-b62f-de11-a6b1-0003ffa70544</ID>
        <Name>Denmark</Name>
```

```
        <PhoneNoFormat>## ## ## ##</PhoneNoFormat>
        <DialingCountryCode>45</DialingCountryCode>
        <InternationalDialingCode>00
        </InternationalDialingCode>
        <InternetTLD>DK</InternetTLD>
    </Country>
    <Country>
        <ID>62750c5e-092c-de11-8c6e-0003ff4ed632</ID>
        <Name>Uganda</Name>
        <PhoneNoFormat></PhoneNoFormat>
        <DialingCountryCode>256</DialingCountryCode>
        <InternationalDialingCode>000
        </InternationalDialingCode>
        <InternetTLD>UG</InternetTLD>
    </Country>
    <Country>
        <ID>63750c5e-092c-de11-8c6e-0003ff4ed632</ID>
        <Name>Great Britain</Name>
        <PhoneNoFormat></PhoneNoFormat>
        <DialingCountryCode>44</DialingCountryCode>
        <InternationalDialingCode>00
        </InternationalDialingCode>
        <InternetTLD>UK</InternetTLD>
    </Country>
    <Country>
        <ID>66750c5e-092c-de11-8c6e-0003ff4ed632</ID>
        <Name>USA</Name>
        <PhoneNoFormat></PhoneNoFormat>
        <DialingCountryCode>1</DialingCountryCode>
        <InternationalDialingCode>011
        </InternationalDialingCode>
        <InternetTLD>US</InternetTLD>
    </Country>
</Countries>
```

Продемонстрируем, как можно добавить, изменить или удалить элемент:

```
// Добавить новый элемент
countries.Add(new XElement("Country",
    new XElement("ID",
                "3032b345-092c-de11-8c6e-0003ff4ed632"),
    new XElement("Name", "Afghanistan"),
```

```
    new XElement("PhoneNoFormat", ""),
    new XElement("DialingCountryCode", "93"),
    new XElement("InternationalDialingCode", "00"),
    new XElement("InternetTLD", "AF")));

// Изменить элемент
// Цикл по всем странам
foreach (XElement country in countries.Elements())
{
    // Этот элемент нужно изменить?
    if (country.Element("Name").Value == "USA")
    {
        // Установить формат телефонного номера
        country.SetElementValue("PhoneNoFormat",
                                    "(###) ###-####");
    }
}

// Удалить элемент
// Цикл по всем странам
foreach (XElement country in countries.Elements())
{
    // Этот элемент нужно удалить?
    if (country.Element("ID").Value ==
                "62750c5e-092c-de11-8c6e-0003ff4ed632")
        // Remove element
        country.Remove();
}
```

В этом примере использование методов Add и Remove в комментариях не нуждается, поскольку это обычные методы для коллекций. Метод SetElementValue ранее нам не встречался, поэтому дадим необходимые пояснения. Если дочерний элемент с именем, полученным через первый параметр, отсутствует, то он будет создан. Если он существовал ранее, его значение будет изменено. Если же в качестве второго параметра указать значение null, то элемент будет удален. Ниже приведен результат работы данного кода: был добавлен Афганистан, была удалена Уганда, был изменен формат телефонных номеров для США.

```
<Countries>
    <Country>
        <ID>602cf8a8-b62f-de11-a6b1-0003ffa70544</ID>
        <Name>Denmark</Name>
        <PhoneNoFormat>## ## ## ##</PhoneNoFormat>
        <DialingCountryCode>45</DialingCountryCode>
        <InternationalDialingCode>00
        </InternationalDialingCode>
        <InternetTLD>DK</InternetTLD>
    </Country>
    <Country>
        <ID>63750c5e-092c-de11-8c6e-0003ff4ed632</ID>
        <Name>Great Britain</Name>
        <PhoneNoFormat></PhoneNoFormat>
        <DialingCountryCode>44</DialingCountryCode>
        <InternationalDialingCode>00
        </InternationalDialingCode>
        <InternetTLD>UK</InternetTLD>
    </Country>
    <Country>
        <ID>66750c5e-092c-de11-8c6e-0003ff4ed632</ID>
        <Name>USA</Name>
        <PhoneNoFormat>(###) ###-####</PhoneNoFormat>
        <DialingCountryCode>1</DialingCountryCode>
        <InternationalDialingCode>011
        </InternationalDialingCode>
        <InternetTLD>US</InternetTLD>
    </Country>
    <Country>
        <ID>3032b345-092c-de11-8c6e-0003ff4ed632</ID>
        <Name>Afghanistan</Name>
        <PhoneNoFormat></PhoneNoFormat>
        <DialingCountryCode>93</DialingCountryCode>
        <InternationalDialingCode>00
        </InternationalDialingCode>
        <InternetTLD>AF</InternetTLD>
    </Country>
</Countries>
```

Отображение данных LINQ to XML

Когда вам требуется отобразить XML-данные на Web-форме, возможны два подхода:

- можно вручную создать XML-дерево в памяти или считать файл, а затем загрузить данные из него в один или несколько Web серверных контролов;
- можно напрямую привязать XML-данные к специальному Web серверному контролу **XmlDataSource**, а затем привязать его к одному или нескольким контролам для показа.

Пример загрузки данных из файла Countries.xml:

```
XElement countries =
        XElement.Load(Server.MapPath("Countries.xml"));
```

Можно также создать XML-документ в памяти в виде строки — пример у нас уже был. Далее нужно добавить на форму контрол **XmlDataSource** (не визуализируется) и привязать к нему данные:

```
// Привязка к строке с документом
CountriesXmlDataSource.Data = xmlString;
// Привязка к файлу с документом
CountriesXmlDataSource.DataFile =
                        Server.MapPath("Countries.xml");
```

Привязку к файлу можно выполнить и декларативно в разметке формы:

```
<asp:XmlDataSource ID="CountriesXmlDataSource"
            runat="server" DataFile="~/Countries.xml" />
```

После привязки данные можно загрузить в контрол и одновременно проверить их на корректность XML-формата:

```
CountriesXmlDataSource.GetXmlDocument();
```

Здесь могут возникнуть проблемы, если данные содержат более одного корневого узла.

Теперь рассмотрим примеры кода для фильтрации элементов и получения всех стран с установленным форматом телефонных номеров. В первом примере мы получаем корневой элемент путем парсинга строки с XML-данными, после чего выбираем нужные страны:

```
// Делаем возможным запрос XML-данных
XElement countries = XElement.Parse(xmlString);
// Получаем страны с установленным форматом
```

```
// телефонных номеров
IEnumerable<XElement> countriesWithPhNoFormat =
    (from c in countries.Elements("Country")
     where c.Attribute("PhoneNoFormat") != null &&
       (string)c.Attribute("PhoneNoFormat").Value != ""
     select c);
```

Объект **countries** для дальнейшего использования можно было бы загрузить в источник типа **XmlDataSource**, воспользовавшись его свойством Data:

```
CountriesXmlDataSource.Data = countries.ToString();
```

Однако если вы хотите загрузить в такой источник страны, отфильтрованные посредством запроса **countriesWithPhNoFormat**, имеющего тип **IEnumerable<XElement>**, запрос надо конвертировать в строку для записи в свойством Data:

```
LoadCountriesXmlDataSource.Data =
    "<Countries>" +
        string.Join("", countriesWithPhNoFormat.Select(
                        x => x.ToString()).ToArray()) +
    "</Countries>";
```

Здесь мы из переменной типа **IEnumerable** с помощью λ-выражения получаем массив строк. Для конкатенации используется метод Join класса **String**. Корневой элемент Countries прописан явно. Если этого не сделать, то метод GetXmlDocument выбросит исключение.

Теперь контрол **XmlDataSource** готов к использованию в паре с data-bound контролом. В следующем примере мы его связываем с контролом типа **GridView** через свойство DataSourceID:

```
// Это декларативно в разметке формы
XmlFileGridView.DataSourceID =
                                "CountriesXmlDataSource";

// А это нужно сделать в коде
XmlFileGridView.DataBind();
```

Кстати, привязку можно было бы сделать и программно, используя свойство data-bound контрола DataSource.

Раздел 3. Управление данными с использованием LINQ to SQL и LINQ to Entities

Данные в реляционной базе данных обычно называют SQL-данными, поскольку доступ к ним осуществляется с использованием синтаксиса языка SQL, точнее, одного из его диалектов. Работая с LINQ to SQL и LINQ to Entities, вы можете употреблять один стандартизованный вариант SQL, даже если имеете дело с базой данных самой различной организации: Microsoft, IBM, Oracle и т.д.

В рамках данного раздела мы узнаем, как использовать LINQ to SQL и LINQ to Entities, как сделать SQL-данные доступными для запроса, как их получить и как управлять ими.

Запрос SQL-данных с использованием LINQ

При работе с ADO.NET вы устанавливаете соединение с реляционной базой данных для их получения или изменения. При использовании LINQ to SQL базовым классом для доступа к данным и их пересылки между сервером и клиентом является класс **System.Data.Linq.DataContext**. LINQ to Entities предоставляет вам для запроса данных и работы с ними как с объектами класс **System.Data.Objects.ObjectContext**.

Далее мы рассмотрим, что в этих вариациях LINQ общего и чем они отличаются друг от друга.

Как сделать SQL-данные доступными для запроса

Прежде всего следует создать соединение с базой данных, для этого удобно использовать Server Explorer. В результате вы получаете доступ к ее таблицам, представлениям, хранимым процедурам и функциям.

Отобразить содержимое базы данных в виде объектов в случае LINQ to SQL вам поможет O/R Designer, а в случае LINQ to Entities — ADO.NET Entity Data Model Designer, также известный как Entity Designer. В первом случае вам потребуется добавить в решение проект LINQ to SQL Classes, во втором — нужен будет проект ADO.NET Entity Data Model. Сделать это можно через диалоговое окно Add New Item.

Замечание. При задании имени проекта лучше сразу выбрать имя, удобное для вас, поскольку имена классов, создаваемых

средой разработки, будут производными от него. Расширения соответствующих файлов: .dbml для O/R Designer и .edmx для Entity Designer.

На поверхности O/R дизайнера есть две панели:

- панель сущностей (слева) отображает классы сущностей, связи и наследование;
- панель методов (справа) отображает методы класса **DataContext**, соответствующие хранимым процедурам и функциям.

Процесс объектно-реляционного отображения (ORM) позволяет вам работать с сущностями базы данных, например с таблицами, как с объектами.

На панели сущностей вы можете в режиме дизайнера создавать классы, задавать связи и наследование. При работе с существующей базой данных на нее можно просто перетаскивать таблицы из окна Server Explorer. При этом, если между таблицами была связь, она будет сохранена и отображена в коде сгенерированных классов.

Замечание. Поддержка O/R Designer в Visual Studio предусмотрена только для SQL Server. Для баз данных других поставщиков класс контекста придется создавать вручную.

А вот какие компоненты нам предлагает Entity Designer:

- поверхность для визуального редактирования концептуальной модели;
- вкладка Mapping Details для просмотра и редактирования отображения;
- окно Model Browser — представление концептуальной модели и модели хранения в виде дерева;
- панель инструментов (Toolbox) для создания сущностей, связей, наследования.

Замечание. Есть набор специальных мастеров (Entity Data Model Wizard, Update Model Wizard, Create Database Wizard) для работы с файлом .edmx. Они позволяют построить модель по существующей базе данных или сгенерировать базу по модели.

Когда вы сохраняете файл .dbml или .edmx, автоматически запускается утилита SqlMetal.exe (в случае LINQ to SQL)

или EntityModelCodeGenerator (в случае LINQ to Entities). В результате создаются (или пересоздаются) классы **DataContext** или **ObjectContext** соответственно.

Замечание. Строка коннекции автоматически сохраняется в файле Web.config в элементе connectionStrings.

Для сравнения кода программы с использованием этих двух вариантов LINQ рассмотрим пару простых примеров. Сначала продемонстрируем код для запроса сущностей из таблицы Customers с использованием LINQ to SQL:

```
// Создаем DataContext, используя
// строку коннекции из Web.config
CustomerManagementDataContext cmDataContext =
    new CustomerManagementDataContext(
    ConfigurationManager.ConnectionStrings[
        "customerManagementConnectionString"].
                                    ConnectionString);

// Получить заказчиков с кредитным лимитом менее 1,000
var query = from c in cmDataContext.Customers
            where c.CreditLimit < 1000
            select c;
// Цикл по заказчикам
    ...
// Освободить DataContext
cmDataContext.Dispose();
```

А теперь то же самое с использованием LINQ to Entities:

```
// Создаем ObjectContext, используя
// строку коннекции из Web.config
CustomerManagementObjectContext cmObjectContext =
    new CustomerManagementObjectContext(
    ConfigurationManager.ConnectionStrings[
        "customerManagementConnectionString"].
                                    ConnectionString);

// Получить заказчиков с кредитным лимитом менее 1,000
var query = from c in cmObjectContext.Customers
            where c.CreditLimit < 1000
            select c;
// Цикл по заказчикам
    ...
```

```
// Освободить ObjectContext
cmObjectContext.Dispose();
```

Обратите внимание на необходимость своевременного освобождения объектов **DataContext** и **ObjectContext**, так как они являются неуправляемыми ресурсами. Также напомним, что LINQ to SQL и LINQ to Entities, точнее, их операторы начинают работу с базой данных, когда вы начинаете получение результатов. Это означает, что объекты **DataContext** и **ObjectContext** нельзя освобождать, пока не получены все данные.

Обработка SQL-данных с использованием LINQ

Разумеется, в случае работы с базой данных требуется не только получать данные оттуда, но и изменять их. Для этого вы также можете использовать объекты **DataContext** и **ObjectContext**. Рассмотрим несколько примеров.

Сначала покажем, как можно изменять данные при использовании LINQ to SQL:

```
// Создать новый объект Country
Country countryObject = new Country
{
    ID = new Guid(
              "3032b345-092c-de11-8c6e-0003ff4ed632"),
    Name = "Afghanistan",
    DialingCountryCode = "93",
    InternationalDialingCode = "00",
    InternetTLD = "AF"
};

// Добавить новую страну в коллекцию Countries
cmDataContext.Countries.InsertOnSubmit(countryObject);

// Запрос на получения объектов для изменения
var modifyQuery =
    from c in cmDataContext.Countries
    where c.Name == "USA"
    select c;
// Изменение объектов
foreach (Country c in modifyQuery)
    c.PhoneNoFormat = "(###) ###-####";
```

```
// Запрос на получения объекта для удаления
var deleteQuery =
    from c in cmDataContext.Countries
    where c.ID == new Guid(
                "62750c5e-092c-de11-8c6e-0003ff4ed632")
    select c;
// Удаление объекта
cmObjectContext.Countries.DeleteOnSubmit(
                                        deleteQuery.First);

// Сохранить изменения в базе данных
try
{
    cmDataContext.SubmitChanges();
}
catch (Exception)
{
    // Обработать исключение
    ...
}
```

Обратите внимание, что для добавления объекта используется метод InsertOnSubmit. Метод DeleteOnSubmit позволяет только отметить объект как подлежащий удалению. Реальное изменение базы данных происходит только при вызове метода SubmitChanges.

А теперь те же самые действия с использованием средств LINQ to Entities:

```
// Создать новый объект Country
Country countryObject = new Country
{
    ID = new Guid(
                "3032b345-092c-de11-8c6e-0003ff4ed632"),
    Name = "Afghanistan",
    DialingCountryCode = "93",
    InternationalDialingCode = "00",
    InternetTLD = "AF"
};

// Добавить новую страну в коллекцию Countries
cmObjectContext.Countries.AddObject(countryObject);
```

```
// Запрос на получения объектов для изменения
var modifyQuery =
    from c in cmObjectContext.Countries
    where c.Name == "USA"
    select c;
// Изменение объектов
foreach (Country c in modifyQuery)
    c.PhoneNoFormat = "(###) ###-####";

// Запрос на получения объекта для удаления
var deleteQuery =
    from c in cmObjectContext.Countries
    where c.ID == new Guid(
                "62750c5e-092c-de11-8c6e-0003ff4ed632")
    select c;
// Удаление объекта
cmObjectContext.Countries.DeleteObject(
                                    deleteQuery.First);

// Сохранить изменения в базе данных
try
{
    cmObjectContext.SaveChanges();;
}
catch (Exception)
{
    // Обработать исключение
    ...
}
```

Как легко заметить, код почти идентичен, отличаются только некоторые названия методов: для добавления страны в коллекцию используется метод AddObject, а для отметки сущности как подлежащей удалению — метод DeleteObject. Изменение самой базы данных производится через вызов метода SaveChanges.

Отображение SQL-данных с использованием LINQ

Давайте теперь разберемся, как можно отобразить наши данные на Web-форме. Конечно, это можно сделать вручную, просто загружая их поштучно в подходящие контролы. Этот

способ вряд ли можно назвать удобным и изящным, хотя в самых простых случаях можно поступать и так.

Более интересным нам представляется вариант с привязкой SQL-данных к Web-серверным контролам **LinqDataSource** или **EntityDataSource** в зависимости от выбранной вариации LINQ. Эти контролы, в свою очередь, привязываются к контролам для отображения данных.

LinqDataSource и **EntityDataSource** оба довольно похожи на **SqlDataSource**, однако можно заметить ряд различий между ними.

LinqDataSource использует для доступа к данным объект **DataContext**, а **EntityDataSource** — **ObjectContext**. SQL-операторы и того и другого могут использовать запросы LINQ. Так что в зависимости от выбранного варианта нам потребуется добавить в свое решение проект LINQ to SQL classes либо ADO.NET Entity Data Model. Затем проект заполняется сущностями и инфраструктура среды разработки генерирует для нас код классов **DataContext** или **ObjectContext**.

Объекты этих классов создаются автоматически при их привязке к контролам **LinqDataSource** или **EntityDataSource**, а также к контролам для показа, таким как **GridView** или **ListView**.

При использовании LINQ to SQL привязка сгенерированного **DataContext** к контролу **LinqDataSource** выглядит примерно так:

```
// Указать класс DataContext
CountriesLinqDataSource.ContextTypeName =
                        "CustomerManagementDataContext";
// Указать таблицу
CountriesLinqDataSource.TableName = "Countries";
```

Отметьте, что, как минимум, требуется установить свойства ContextTypeName и TableName. Впрочем, это можно сделать и прямо в разметке формы:

```
<asp:LinqDataSource ID="CountriesLinqDataSource"
    runat="server"
    ContextTypeName="CustomerManagementDataContext"
    TableName="Countries" />
```

При использовании LINQ to Entities привязка сгенерированного **ObjectContext** к контролу **EntityDataSource** могла бы выглядеть так:

```
// Указать класс ObjectContext
CountriesEntityDataSource.ContextTypeName =
                        "CustomerManagementObjectContext";
// Указать сущность
CountriesEntityDataSource.EntitySetName = "Countries";
```

А это вариант их декларативной установки:

```
<asp:EntityDataSource ID="CountriesEntityDataSource"
    runat="server"
    ContextTypeName="CustomerManagementObjectContext"
    EntitySetName="Countries" />
```

Теперь вы можете отобразить строки таблицы с использованием экземпляра класса, соответствующего ей (**DataContext** или **ObjectContext**).

Пока мы не заметили каких-либо различий в возможностях использования этих двух типов источников от **SqlDataSource**.

По-настоящему полезными классы **LinqDataSource** и **EntityDataSource** делает свойство Where и связанные с ним параметры. Они могут быть установлены в значения, полученные из самых разных источников: параметров контрола, параметров куки, параметров формы, параметров профиля, параметров строки запроса, параметров сеанса — да и просто могут быть заданы как константы.

Это означает, что вы можете фильтровать данные автоматически, основываясь на этих значениях (например, полученных из поля ввода).

Вы можете сконфигурировать свойство Where вручную. Проще всего использовать для этого диалоговое окно Configure Where Expression (оно доступно через окно Properties). Можно это сделать и в диалоговом окне Configure Data Source, которое доступно на Web-форме в режиме дизайнера после того, как на ней разместили **LinqDataSource** или **EntityDataSource**.

Ниже приведен пример того, как могла бы выглядеть разметка **LinqDataSource**:

```
<asp:LinqDataSource ID="CountriesLinqDataSource"
    runat="server"
    ContextTypeName="CustomerManagementDataContext"
    TableName="Countries" EnableDelete="True"
    EnableInsert="True" EnableUpdate="True"
    Where="Name == @Name">
```

```
    <WhereParameters>
        <asp:Parameter DefaultValue="B" Name="Name"
                       Type="String" />
    </WhereParameters>
</asp:LinqDataSource>
```

Лекция 10. Управление данными с использованием ASP.NET Dynamic Data

Рассматриваемая здесь технология Dynamic Data относится к категории средств быстрой разработки — RAD (Rapid Application Development), которые позволяют очень быстро создать прототип приложения. Затем вы можете перетащить из окна инструментов элементы управления на поверхность дизайнера, после чего собрать и запустить свою программу. Благодаря такому подходу вы можете очень оперативно обсуждать будущее приложение с заказчиком или с коллегами, если это коллективная разработка, и быстро вносить изменения.

Вместе с тем использование RAD имеет свои особенности. Качество генерируемого автоматически кода может быть очень разным, и этот код не всегда легко поддерживать. Кроме того, существуют требования безопасности, удовлетворять которые придется путем ручного кодирования. Это в полной мере относится и к Web-приложениям, которые позволяют не только просматривать данные, но и изменять их. Впрочем, технология Microsoft ASP.NET Dynamic Data позволяет решать эти проблемы в сочетании с RAD-разработкой.

Dynamic Data можно воспринимать как оболочку для быстрой разработки приложений, управляемых данными (data-driven), основанных на модели LINQ to SQL или Entity Framework. Эти приложения могут использовать, модифицировать и анализировать данные для выполнения разных задач. Они поддерживают маршрутизацию при обращении к источнику данных. Их можно использовать, например, для создания Web-сайтов, которые управляют данными, автоматически их читая или записывая в источник, и при этом совсем (или почти) не требуют ручного кодирования.

Раздел 1. Обзор ASP.NET Dynamic Data

ASP.NET Dynamic Data — это своего рода пристройка к системе быстрой разработки приложений. Она обеспечивает функционал Web-приложений для просмотра и редактирования данных.

Если вы добавляете в свое приложение модель LINQ to SQL или Entity Framework, вы можете зарегистрировать ее с Dynamic Data. В результате вы получите полнофункциональный Web-сайт для управления вашими данными. При этом он будет иметь возможность выполнять весь набор операций, включая фильтрацию данных по внешним ключам и булевским полям. Также вы можете добавить проверку корректности, основанную на ограничениях базы данных на нулевые поля, типы данных, длину полей с использованием знакомых контролов валидации.

Как работает ASP.NET Dynamic Data?

Как было сказано, надстройка ASP.NET Dynamic Data помогает нам создавать управляемые данными Web-приложения. Оболочка Dynamic Data автоматически обнаруживает метаданные модели данных во время исполнения и использует их для определения поведения пользовательского интерфейса.

Надстройка предоставляет функционал для управления данными. Вы можете с ее помощью настроить ее поведение по умолчанию. Также вы можете интегрировать эти дополнительные элементы с существующими Web-формами и приложениями.

Модель данных представляет информацию в базе данных и отображает, как объекты базы связаны друг с другом. Вы можете зарегистрировать модель данных LINQ to SQL или ADO.NET Entity Framework вместе с Dynamic Data. В результате она сможет выполнять автоматическую валидацию полей данных. Также вы сможете контролировать появление и поведение данных через объявление и применение атрибутов модели данных. Например, вы можете объявить таблицу частью Web-приложения и указать поля, которые должны быть показаны при демонстрации таблицы. При этом имя поля вы можете задать не такое, как в таблице; можно добавить еще атрибут для описания поля в виде всплывающей подсказки, которая появляется при наведении на него курсора мыши.

Вы можете применять в Web-приложении разные модели данных, но те, которые используются с Dynamic Data, должны иметь один тип.

Формирование шаблонов обеспечивает нам следующие дополнительные возможности:

- позволяет создавать управляемые данными Web-приложения с минимальным объемом ручного кодирования, а то и вовсе без него;
- обеспечивает встроенную валидацию, основанную на схеме базы данных, которая является частью схемы модели данных;
- создает автоматические фильтры для каждого внешнего ключа или булевского поля, которые определены в схеме базы данных.

Шаблоны

Оболочка ASP.NET Dynamic Data основана на множестве обобщенных шаблонов, которые вы можете использовать для отображения различных данных. Они хранятся в виде файлов на диске. Такие шаблоны, как Web-формы и шаблоны пользовательских контролов, используются Dynamic Data на этапе выполнения для генерации страниц, содержащих данные из источника, а также для управления этими страницами. Нам предоставляются следующие шаблоны.

- *Шаблоны страниц*. Они используются для задания представления сущностей данных по умолчанию. Фактически это Web-формы, сконфигурированные для показа данных из сущностей конкретной модели данных. Они обобщены, поскольку Web-формы не привязаны к конкретной сущности базы данных, например к таблице. Dynamic Data предоставляет нам набор шаблонов страниц для различного представления данных, включая:
    - *List view* — представление по умолчанию для строк таблицы;
    - *Details view* — представление для отдельных данных из строки;
    - *Edit view* — представление для отдельных данных из строки, допускающее их редактирование;
    - *List Details view* — отображение родительской и дочерней сущности в стиле " master/details";

- *Insert view* — представление для добавляемой строки.

Вы можете для своего Web-приложения сделать любой из шаблонов шаблоном по умолчанию. Также можно использовать разные шаблоны для разных страниц или на основе имеющегося шаблона создать свой.

- *Шаблоны полей*. Они используются для создания интерфейса пользователя и управления данными в отдельных полях. Dynamic Data выбирает подходящий шаблон, исходя из типа данных, прописанного в схеме модели данных. Dynamic Data имеет отдельные шаблоны для отображения и для редактирования полей данных.

  Вы можете изменить шаблоны полей, используемых по умолчанию и создать собственную модель данных, которая использует ваши шаблоны. Это делается путем применения атрибута UIHint к полю, использующему шаблон.

Контролы для отображения данных

В ASP.NET есть несколько сложных, основанных на шаблонах контролов из этой категории, например: **GridView**, **DetailsView**, **ListView**, **FormView**. Dynamic Data расширяет их возможности, добавляя динамическое поведение. Рассмотрим их основные характеристики.

- **GridView** и **DetailsView** отображают данные динамически, используя предопределенные шаблоны Dynamic Data. Вы можете перестроить эти шаблоны или воспользоваться специальными элементами управления, создавая пользовательский интерфейс для управления полями данных. Сделанные в одном месте изменения приведут к изменению поведения и отображения во всем Web-приложении. Для отображения значения в контролах **GridView** и **DetailsView** используется класс **DynamicField**.
- **ListView** и **FormView** похожи на них, но реализуют свое поведение через использование контрола **DynamicControl**. В отличие от **GridView** и **DetailsView** вы должны вручную указать поле сущности данных, которой **ListView** и **FormView** будут управлять.

Во время выполнения Dynamic Data автоматически создает пользовательский интерфейс для этих контролах, основываясь на шаблонах. Но контрол **DynamicControl** не визуализирует автоматически никакие поля, а значит, вы должны сами связать эти контролы с конкретными полями данных.

Также во время выполнения программы Dynamic Data исследует метаданные модели данных и обеспечивает автоматическую валидацию на их основе. Например, если в столбце не допускается пустое значение, автоматически будет добавлен контрол **RequiredFieldValidator**. Вы можете применять и собственные метаданные, чтобы настроить этот процесс.

Устройство проекта Dynamic Data

Инфраструктура ASP.NET Dynamic Data Web-сайта или Web-приложения несколько отличается от таковой для стандартного Web-сайта или приложения. Она включает в себя несколько дополнительных папок и файлов.

При создании заготовки Web-сайта или приложения с использованием Dynamic Data соответствующие мастера по заготовкам генерируют пользовательские контролы и шаблоны страниц, а уже механизмы Dynamic Data окончательно создают интерфейс пользователя для работы с данными.

Такие проекты требуют наличия объекта класса **MetaModel** из пространства имен **System.Web.DynamicData**, а также зарегистрированных классов **DataContext** или **ObjectContext**. Последние должны располагаться в подпапке App_Code корневой папки приложения.

Папки и файлы

Перечислим основные подпапки и файлы из корневой папки Web-приложения с использованием Dynamic Data.

- *DynamicData*. Подпапка содержит пользовательские контролы и Web-формы для отображения данных. Пользовательские контролы задают шаблоны полей, а Web-формы — шаблоны страниц.
- *Global.asax*. Файл присутствует и в стандартных Web-приложениях. В приложениях с использованием Dynamic Data в нем регистрируется экземпляр класса **MetaModel**, а также **DataContext** или **ObjectContext** вместе с ним.

Здесь также выполняется добавление маршрутов в объект класса **System.Web.Routing.RouteCollection**.

- *Scripts*. Подпапка содержит файлы с кодом на языке JavaScript для использования возможностей технологий Ajax (Asynchronous JavaScript and XML) и jQuery.
- *Site.master*. Мастер-страница в приложениях с использованием Dynamic Data похожа на мастер-страницу в стандартных Web-приложениях. Дополнительно она содержит контрол **System.Web.UI.ScriptManager**, в котором свойство EnablePartialRendering установлено в значение false. Класс **ScriptManager** необходим для использования в Ajax -приложениях.
- *Web.config*. Файл конфигурации, такой же, как и в стандартных Web-приложениях.

Файл Global.asax

Файл Global.asax, как обычно, содержит код обработчиков событий уровня страницы, например события Application_Start. В случае Dynamic Data этот обработчик содержит вызов метода RegisterRoutes. В его коде по умолчанию содержится экземпляр класса **MetaModel** и вызов метода RegisterContext в виде комментария. Чтобы сделать возможными операции Dynamic Data, комментарий с вызова нужно снять и передать ему правильный контекст данных (экземпляр **DataContext** или контекст ADO.NET Entity Framework).

По умолчанию в случае Dynamic Data вызывается метод Add для добавления маршрута каждого действия, а также для страниц List.aspx, Details.aspx, Edit.aspx и Insert.aspx

Структура папки DynamicData

Папка содержит следующие подпапки.

- *Content*. Папка содержит подпапку Images для хранения изображений, используемых в качестве иконок для пользовательского контрола **GridViewPager**, а также код этого контрола в файле GridViewPager.ascx. Он используется для улучшения переключения страниц, если таковых несколько для данной сущности.

- *CustomPages*. Папка играет роль контейнера для заказных шаблонов страниц, заменяющих собой шаблоны по умолчанию из папки DynamicData\PageTemplates.
- *EntityTemplates*. Папка содержит шаблоны сущностей по умолчанию, которые используются при создании пользовательского интерфейса для отображения таблиц.
- *FieldTemplates*. Папка содержит пользовательские контролы для показа и редактирования данных типов, выведенных из модели данных, таких как Boolean.ascx, Boolean_Edit.ascx, DateTime.ascx, DateTime_Edit.ascx и многие другие.
- *Filters*. Папка содержит заказные шаблоны полей, использующихся для фильтрации выводимых строк таблицы, такие как Boolean.ascx, Enumeration.ascx и ForeignKey.ascx (пользовательские контролы).
- *PageTemplates*. Папка содержит встроенные шаблоны страниц Details.aspx, Edit.aspx, Insert.aspx, List.aspx и ListDetails.aspx, которые используются при создании пользовательского интерфейса для управления данными.

Динамическое построение страниц

Процесс динамического построения страниц (scaffolding) во время выполнения программы основан на модели данных и не требует наличия физической страницы.

Для того чтобы управлять всеми данными в рамках модели, такое конструирование должно быть доступно для всех сущностей, а не только для некоторых. Но если вы делаете возможной такую надстройку для всех данных, то пользователь, хочет он этого или нет, получает доступ к схеме базы данных в полном объеме, что не всегда безопасно. Впрочем, если вы разрабатываете совсем простое приложение, такой подход может оказаться оправданным, поскольку он экономит время на разработку.

Включить этот режим можно при регистрации контекста данных или объекта в Global.asax. Для этого надо вызвать метод RegisterContext, который принимает в качестве параметра объект **System.Web.DynamicData.ContextConfiguration**. Впрочем, при использовании Dynamic Data это нужно будет сделать, даже если вы не собираетесь использовать механизм надстраивания.

Для включения этого механизма свойство ScaffoldAllTables объекта **ContextConfiguration** следует установить в значение true:

```
DefaultModel.RegisterContext(typeof(MyObjectContext),
    new ContextConfiguration()
                        { ScaffoldAllTables = true });
```

Риск здесь заключается в том, что таким образом вы все таблицы выставляете напоказ. Более того, если потом вы добавите к классу **DataContext** или **ObjectContext** новые сущности, они также будут доступны для надстраивания.

Впрочем, даже если вы выбрали такой подход, вы можете сознательно спрятать сущности от механизма надстраивания, используя атрибут ScaffoldTableAttribute. В этом случае вы можете свойство ScaffoldAllTables при регистрации модели данных установить в значение false. Этот атрибут можно использовать, чтобы разрешить или запретить применение надстраивания к заданной сущности.

Атрибут применяется к частичному классу с тем же именем, что и у сущности в модели данных:

```
using System.ComponentModel.DataAnnotations;
    ...
[ScaffoldTable(true)]
public partial class Order
{
    ...
}
```

Обратите внимание на использованное пространство имен, а также на употребление единственного и множественного числа. Здесь оно единственное, так как должно совпадать с именем класса в LINQ to SQL или модели ADO.NET Entity Data.

Шаблоны ASP.NET Dynamic Data

Как мы уже говорили, Dynamic Data при генерации пользовательского интерфейса для управления данными использует шаблоны страниц и шаблоны полей. Создание результирующих страниц и элементов управления происходит на этапе выполнения программы. Шаблоны содержат контролы из категории data-bound и имеют различные настройки свойств по умолчанию, которые используются для автоматического связывания шаблонов с источником данных.

Шаблоны страниц

Эти встроенные шаблоны могут использоваться для создания полнофункциональных страниц. Они позволяют выполнять все операции над данными: просмотр, вставку, редактирование, удаление, а также фильтрацию и постраничный просмотр (шаблон List view из файла List.aspx).

Кроме того, обеспечивается валидация значений, основанная на схеме базы данных, и сортирующие фильтры по каждому внешнему ключу или булевскому полю. Разумеется, все шаблоны имеют общий характер и не привязаны к конкретной сущности.

Dynamic Data использует разные шаблоны страниц для разных операций. Исключением является операция удаления, которая отображается как ссылка на странице List и на странице Details, так как нет необходимости иметь для нее отдельную страницу — достаточно диалога для подтверждения операции.

Перечислим имена файлов, определяющих обычный набор шаблонов, без комментариев, поскольку их имена достаточно информативны: List.aspx, Details.aspx, Edit.aspx, ListDetails.aspx, Insert.aspx.

Шаблоны полей

Они используются при создании интерфейса пользователя для управления данными в отдельных полях. Dynamic Data выбирает подходящий шаблон, исходя из типа значений, прописанного в схеме базы данных. Реализованы шаблоны полей в виде пользовательских контролов, соответствующие файлы имеют расширение .ascx.

Dynamic Data считывает схему базы данных, чтобы получить информацию о типах значений и внешних ключах, и подбирает подходящие контролы для отображения данных. Это делает доступ к данным и управление ими очень простым.

Если вас не устраивает стандартное отображение данных в контролах **GridView** или **FormView**, без Dynamic Data вам пришлось бы вручную создать код для отображения каждой страницы. Но наличие шаблонов полей обеспечивает простой способ внесения изменений во внешний вид этих контролов по умолчанию.

Мы не будем здесь перечислять и комментировать состав довольно обширного набора стандартных шаблонов полей: в про-

екте Visual Studio их легко найти, а их названия говорят сами за себя.

Маршрутизация ASP.NET Dynamic Data

Dynamic Data использует механизмы навигации ASP.NET для исполнения URL-запросов. Маршрутизация определяется в глобальном файле приложения Global.asax. Механизм надстраивания догадывается о сущности, которую хочет видеть пользователь, и ее представлении по URL-запросу.

Преимущество использования механизма маршрутизации состоит в том, что запрошенный URL не обязан соответствовать пути в приложении, причем маршрутизация разрешена и для физических, и для динамически созданных страниц.

Вы можете настроить этот механизм так, чтобы использовать любой шаблон страницы в качестве страницы по умолчанию для конкретной операции или использовать разные шаблоны страниц для разных целей. Вы можете также настроить маршруты для разных шаблонов страниц, скажем на показ URL, или удалить расширение файла (.aspx), или передать параметры через механизм маршрутизации вместо строки запроса.

Приведем в качестве примера кусочек кода по умолчанию из файла Global.asax:

```
routes.Add(new
DynamicDataRoute("{table}/{action}.aspx")
    { Constraints = new RouteValueDictionary(new
                { action = "List|Details|Edit|Insert" }),
    Model = DefaultModel
});
```

Здесь определяется маршрут по умолчанию. Шаблон задается для каждого действия на каждой странице.

Следующий пример показывает, как можно задать один шаблон ListDetails для двух операций: List и Details. Для удобства разработчиков исходный код из Global.asax содержит образцы такого рода операторов в виде комментариев.

```
routes.Add(new
    DynamicDataRoute("{table}/ListDetails.aspx")
    {
        Action = PageAction.List,
        ViewName = "ListDetails",
```

```
        Model = DefaultModel
    });

routes.Add(new
    DynamicDataRoute("{table}/ListDetails.aspx")
    {
        Action = PageAction.Details,
        ViewName = "ListDetails",
        Model = DefaultModel
    })
```

Маршруты анализируются в порядке их появления в коде. Поэтому следует сначала определять более специфические маршруты, а уже затем более общие. Вы можете указать отдельные маршруты для конкретных таблиц через указание шаблона страницы, отличного от шаблона для оставшихся страниц.

Следующий пример показывает, как это сделать для таблицы Customers. Второй вызов назначает общий шаблон для всех оставшихся страниц.

```
routes.Add(new
    DynamicDataRoute("Customers/{action}.aspx")
    {
        ViewName = "ListDetails",
        Table = "Customers",
        Model = model
    });

routes.Add(new
    DynamicDataRoute("{table}/{action}.aspx")
    {
        Constraints = new RouteValueDictionary(
        new { action = "List|Details|Edit|Insert" }),
        Model = model
    });
```

Раздел 2. Применение ASP.NET Dynamic Data

Использовать технологию ASP.NET Dynamic Data можно по-разному: можно сразу создать новый Web-сайт, основанный на шаблоне ASP.NET Dynamic Data Web Site, но можно и добавить функциональность Dynamic Data к существующему сайту.

Но ваши возможности этим не ограничиваются, поскольку вы еще можете перенастроить уже существующие или создать новые шаблоны страниц. Кроме того, вы можете перенастроить шаблон поля, а также добавить динамическое поведение к data-bound контролу.

Как это все можно проделать, мы рассмотрим довольно схематично, поскольку лучше эти действия показывать на компьютере, а кроме того, в разных версиях Microsoft Visual Studio они могут выглядеть несколько по-разному. Поэтому мы ограничимся перечислением основных шагов.

Создание нового Web-сайта ASP.NET Dynamic Data

1. Через меню **File** создаем новый Web-сайт по шаблону **ASP.NET Dynamic Data Entities Web Site**.
2. Подключаем источник данных. Пусть в нашем случае это будет файл AdventureWorksLT2008_Data.mdf. При использовании в качестве источника данных именно файла его лучше скопировать в подпапку App_Data в корневой папке приложения.
3. В Solution Explorer через контекстное меню **Add New Item** добавляем к проекту модель данных **ADO.NET Entity Data Model**, назвав ее, например, AdventureWorks.edmx. В диалоге подключения модели выберем вариант Generate from database и укажем наш файл с данными. В следующих окнах диалога укажем таблицы, которые мы хотели бы включить в модель.
4. Откроем файл Global.asax и добавим регистрацию контекста в методе RegisterRoutes, например, так:

```
DefaultModel.RegisterContext(
    typeof(AdventureWorksLT2008_DataModel.
                    AdventureWorksLT2008_DataEntities),
          new ContextConfiguration()
{
    ScaffoldAllTables = true
});
```

5. Теперь проект можно компилировать и просматривать в браузере, ну и, разумеется, дорабатывать и настраивать по своему усмотрению.

Добавление возможностей Dynamic Data к существующему Web-сайту

Это уже более трудоемкий процесс, заставляющий нас пожалеть, что мы не запланировали такие возможности сразу. Что нам потребуется сделать?

1. Создаем отдельный проект, который послужит нам в качестве образца Web-сайта с возможностями Dynamic Data.
2. Копируем разметку и код из образца в ваш ранее созданный Web-сайт.
3. Копируем в наш сайт недостающие файлы и папки.
   3.1. Подпапку DynamicData из корневой папки со всем ее наполнением по умолчанию.
   3.2. В Solution Explorer обновляем показ содержимого Web-сайта и убеждаемся, что папка DynamicData добавлена.
4. Добавляем мастер-страницу или модифицируем существующую, если она уже есть.
   4.1. Добавляем мастер-страницу, если ее еще нет.
   4.2. Шаблоны страниц Dynamic Data ссылаются на файл Site.master. Вы должны или переименовать свою мастер-страницу, используя это имя, или изменить каждый шаблон страницы в папке DynamicData\PageTemplates так, чтобы они ссылались на правильную мастер-страницу.
5. Добавляем контрол **ScriptManager**. Это нужно, потому что страницы Dynamic Data используют контролы Ajax , в частности контрол **UpdatePanel**. Поэтому нужно или добавить **ScriptManager** на мастер-страницу, или изменить существующую.
   5.1. Открываем мастер-страницу.
   5.2. Если нужно, добавляем **ScriptManager** перетаскиванием с панели инструментов:

```
<asp:ScriptManager ID="ScriptManager1"
                   runat="server" />
```

Его имя должно быть именно **ScriptManager1**, поскольку контрол ScriptManagerProxy на каждой шаблонной странице ссылается именно на него.

5.3. Сохраняем мастер-страницу.

6. Добавляем модель данных LINQ to SQL или Entity Framework data model. Для этого, как и ранее, нужно будет добавить в решение проект подходящего типа и добавить туда таблицы из окна Server Explorer.

7. Если в проекте отсутствует файл Global.asax, добавляем новый элемент Global Application Class project.

7.1. Добавляем в него директиву using для пространства имен System.Web.DynamicData и создаем заготовку для метода Register по образцу:

```
public static void Register(RouteCollection routes)
{
}
```

7.2. В метод Application_Start добавляем вызов метода Register по образцу:

```
Register(RouteTable.Routes);
```

7.3. В файле Global.asax в самом начале блока скриптов создаем и инициализируем закрытый статический экземпляр класса **System.Web.DynamicData.MetaModel**:

```
private static MetaModel s_defaultModel =
                                        new MetaModel();
```

7.4. В Global.asax добавляем свойство DefaultModel, возвращающее этот экземпляр **MetaModel**:

```
public static MetaModel DefaultModel {
    get {
          return s_defaultModel;
    }
}
```

7.5. В процедуре Register регистрируем класс **DataContext** или **ObjectContext**, отключив надстройку для всех таблиц:

```
DefaultModel.RegisterContext(typeof(MyContext),
                             new ContextConfiguration()
{
    ScaffoldAllTables = false
});
```

7.6. В классе **Global Application** после директивы Application добавляем директиву Import с атрибутом Namespace, установленным в System.Web.Routing.

7.7. В процедуре Register добавляем обобщенный маршрут типа **DynamicDataRoute** для операций List, Details, Edit и Insert именно в таком порядке. Наличие маршрута приводит к перенаправлению на шаблон страницы с таким именем, в качестве префикса в URL используется имя таблицы. Добавление маршрута производится через вызов метода Add с передачей параметра **routes**:

```
routes.Add(new
                DynamicDataRoute("{table}/{action}.aspx")
{
    Constraints = new RouteValueDictionary(
        new
        {
            action = "List|Details|Edit|Insert"
        }),
        Model = DefaultModel
});
}
```

7.8. Сохраняем и закрываем Global.asax.

Добавление динамического поведения контролам для привязки данных

ASP.NET Dynamic Data содержит несколько классов, которые вы можете использовать для добавления динамического поведения контролам ASP.NET. Эти классы, в числе которых можно назвать **DynamicDataManager**, **DynamicControl** и **DynamicField**, собраны в пространстве имен System.Web.DynamicData.

Вы можете добавить функциональность Dynamic Data элементам управления **FormView** или **GridView**, используя в режиме дизайнера поле **DynamicField**. Если требуется добавить объекты **DynamicField** к data-bound контролам через использование дизайнера, на форме или пользовательском контроле обязательно должен быть контрол **DynamicDataManager**.

Для добавления объектов **DynamicField** к data-bound контролам нужно выполнить следующие действия.

1. Открыть Web-форму в режиме дизайнера.
2. Развернуть в панели инструментов раздел Dynamic Data и перетащить контрол **DynamicDataManager** на форму.
3. В контекстном меню data-bound контрола выбрать пункт **Show Smart Tag**.
4. В появившейся панели выбрать пункт **Edit Columns**.
5. В диалоговом окне Fields снять флажок Auto-generate fields.
6. В списке Available fields щелкнуть на пункте **DynamicField**, а затем **Add**.
7. В списке Field properties указать в свойстве DataField имя столбца, к которому мы хотим привязать динамическое поле.

Раздел 3. Настройка ASP.NET Dynamic Data

ASP.NET Dynamic Data предоставляет вам набор встроенных шаблоном страниц и полей. Но, возможно, вы захотите изменить их внешний вид, выбрать другие контролы для полей или изменить процесс надстраивания. Например, вам может потребоваться создать шаблон поля для ввода и редактирования телефонного номера или адреса электронной почты в специфическом формате.

Данный раздел посвящен вопросам перенастройки существующих или создания новых шаблонов полей, а также перенастройки процесса надстраивания и маршрутизации для Web-сайтов с поддержкой технологии Dynamic Data.

Создание шаблона страницы

Dynamic Data позволяет вам создавать заказные шаблоны страниц путем перестройки встроенных, а также путем подгонки размещения отдельных ее элементов. Шаблоны страниц в папке DynamicData\PageTemplates обобщены и используются для всех сущностей в вашей модели данных, пока вы не создадите заказной шаблон страницы или не измените процесс надстраивания.

Если вы хотите настроить один или более шаблонов страниц для всех сущностей в вашей модели данных, вы можете изменить эти шаблоны прямо в папке DynamicData\PageTemplates. Находящиеся там изначально шаблоны уже имеют наполнение, и для их подгонки могут потребоваться дизайнеры и/или разработчики.

Изменения, которые вы там сделаете, отразятся на внешнем виде и поведении всех страниц, генерируемых на основе данного шаблона с использованием механизма надстраивания. Также вы можете добавить в эту папку новый шаблон страницы, а затем перестроить механизм надстраивания так, чтобы использовать этот шаблон для одного или более действий.

Если же вы хотите создать свое представление для конкретной сущности в вашей модели данных, вы можете использовать заказной шаблон страницы. Для этого потребуется выполнить следующую последовательность шагов.

1. Открыть свой Web-сайт в Visual Studio.
2. В окне Solution Explorer развернуть папку DynamicData и создать в ней папку в подпапке CustomPages.
3. Назвать созданную папку по имени соответствующей сущности в вашей модели данных во множественном числе; например, для сущности Order она должна называться Orders.
4. Создайте новый шаблон страниц с нуля или просто скопируйте существующий шаблон страницы из папки DynamicData\PageTemplates в новую подпапку.
5. Настройте шаблон страницы по своему усмотрению и сохраните изменения.
6. Протестируйте свое приложение и новый шаблон страницы, выполнив действие, для которого создавался данный шаблон.

Замечание. Если имя подпапки для заказного шаблона не совпадает с именем сущности во множественном числе, Dynamic Data не сможет найти шаблон страницы и будет использовать один из обобщенных шаблонов из папки PageTemplates.

Настройка механизма скаффолдинга

Мы уже знаем, как включить или отключить применение механизма надстраивания для всех сущностей через задание значения свойства ScaffoldAllTables класса **ContextConfiguration**.

Для более тонкой настройки этого механизма предназначены атрибуты ScaffoldTableAttribute и ScaffoldColumnAttribute. Для их использования потребуется подключить необходимое пространство имен:

```
using System.ComponentModel.DataAnnotations;
```

Атрибут ScaffoldTableAttribute используется для большего контроля за тем, какие таблицы делаются доступными. Чтобы его использовать вы должны создать частичный класс с тем же именем, что и у сущности в модели данных, и применить к нему этот атрибут, например:

```
[ScaffoldTable(true)]
public partial class Order
{
}
```

Если вам нужен больший контроль за тем, какие поля выставляются, для включения или отключения этого механизма можно использовать атрибут ScaffoldColumnAttribute. Перечислим правила его использования.

- По умолчанию все поля показываются.
- Если к полю применен атрибут UIHintAttribute, оно показывается.
- Если в поле содержится внешний ключ, оно не показывается. Причина в том, что Dynamic Data работает с внешними ключами иначе, чем с другими значениями.
- Если поле автоматически генерируется в базе данных, его значение не показывается (при отсутствии атрибута UIHintAttribute). Причина заключается в том, что поле может не содержать значимой информации.
- Если значение свойства IsCustomProperty установлено в значение true, поле не показывается.

Чтобы применить атрибут ScaffoldColumnAttribute, нужно создать соответствующий класс метаданных, в котором вы припишете этот атрибут к полю данных. Кроме того, надо создать частичный класс с таким же именем, как у класса сущности, и к этому классу применить атрибут MetadataTypeAttribute. Приведем пример того, как можно спрятать поля ID и Address:

```
[MetadataType(typeof(CustomerMetadata))]
public partial class Customer
{
}
```

```
public class CustomerMetadata
{
    [ScaffoldColumn(false)]
    public object ID;
     [ScaffoldColumn(false)]
    public object Address;
}
```

Лекция 11. Использование Ajax в ASP.NET

Ajax (Asynchronous JavaScript and XML) — это набор технологий, позволяющих сделать ваши Web-страницы более поворотливыми.

Реализация Ajax от Microsoft (Microsoft Ajax) включает в себя ряд таких технологий: частично-страничную отправку, асинхронные обращения к Web-серверу и использование специализированных элементов интерфейса.

Раздел 1. Введение в Ajax

Используя Ajax, Web-приложение может получать данные с сервера асинхронно в фоновом режиме без отображения на дисплее и изменения существующей страницы.

Microsoft Ajax помогает вам создавать Web-страницы, которые поддерживают частичное обновление страницы — только те области, которые требуется обновить. В сочетании с поддержкой асинхронных вызовов и другими возможностями это позволяет сократить трафик и ускорить загрузку страниц.

Что такое Asynchronous JavaScript and XML?

Ajax работает на многих платформах и браузерах, он использует открытые стандарты, такие как JavaScript. Асинхронная природа Ajax позволяет вашему приложению получать данные с сервера в фоновом режиме и не влияет на отображение текущей Web-страницы.

Есть множество типов информации, которая может обновляться через Ajax на Web-сайтах: спортивные результаты, цены, новости, время и многое другое. Обычно ее обновляют через определенные интервалы времени с использованием таймера.

Ajax — это не технология, а скорее множество технологий и инструментов. Перечислим некоторые из них.

- Объект **XMLHttpRequest**. Его можно использовать для асинхронного обмена информацией между браузером и Web-сервером. Этот объект позволяет запрашивать именно данные, а не страницу целиком, и это ключевая черта Ajax.

- Языки HTML, XHTML и каскадные таблицы стилей (CSS), которые обеспечивают разметку и стиль представления информации на Web-странице.
- XML и другие форматы передачи данных, такие как JSON (JavaScript Object Notation), которые можно использовать для обмена данными между Web-сервером и клиентом.
- DOM (Document Object Model), которая используется вместе со скриптовым языком (например, JavaScript) для облегчения взаимодействия с данными Web-страницы на стороне клиента.

Как устроен ASP.NET Ajax?

ASP.NET Ajax позволяет разрабатывать Web-страницы с одновременным использованием технологий ASP.NET и Ajax. Его можно представлять себе как оболочку для создания интерактивных и шустрых Web-приложений, которые предназначены для работы с большинством популярных браузеров.

Перечислим основные компоненты этой оболочки.

- *Ajax серверной стороны*. Включает в себя ASP.NET версии 4.0 или выше для Web-форм, а также их поддержку и набор серверных контролов, в частности серверные контролы для добавления Ajax-функциональности ASP.NET Web-формам.
- *Ajax клиентской стороны*. Это Microsoft Ajax Library — библиотека скриптов на языке JavaScript, обеспечивающих функциональность клиентской стороны. Этот компонент выпускается отдельно от ASP.NET Framework или Microsoft Visual Studio, однако он легко встраивается в среду разработки.
- *Ajax Control Toolkit*. Это мощный инструмент разработки, позволяющий легко создавать и использовать заказные контролы и так называемые экстендеры (элементы управления для расширения функциональности других контролов). Он включает в себя не только готовые элементы, но и образцы и компоненты, чтобы вы могли

также их создавать. Этот набор поддерживается и расширяется сообществом программистов.

- *Библиотека jQuery*. Это библиотека с кодом на языке JavaScript для запросов и манипуляций данными на базе модели DOM.
- *CDN (Microsoft Ajax Content Delivery Network)*. Сеть позволяет вам легко добавлять в Web-приложения скрипты Microsoft Ajax Library и jQuery. Используя ее, можно значительно улучшить производительность Ajax-приложений, поскольку содержимое CDN закэшировано на множестве серверов по всему миру и легко может быть считано браузером.

ASP.NET Ajax помогает программистам выбрать подходящий метод Ajax-разработки. Это относится к программированию и серверной, и клиентской стороны и их комбинации. ASP.NET Ajax интегрирован в .NET Framework версии 4 и более новые, он позволяет создавать Ajax-приложения с использованием средств, предлагаемых Visual Studio.

Перечислим преимущества Ajax-приложений по сравнению с обычными Web-приложениями:

- бóльшая эффективность, поскольку значительная часть обработки выполняется в браузере;
- удобные готовые элементы пользовательского интерфейса: индикаторы прогресса, всплывающие подсказки, календари и огромное множество других;
- частично-страничная отправка, позволяющая сократить трафик и ускорить работу;
- возможность использования служб ASP.NET для обеспечения безопасности: формы аутентификации, роли, пользовательские профили;
- автоматически генерируемые классы proxy, которые облегчают вызов методов Web-служб из клиентских скриптов;
- оболочка, которая позволяет переделывать серверные контролы под потребности клиента;
- поддержка для большинства популярных и часто используемых браузеров;
- интеграция с данными посредством Web-служб.

Архитектура компонентов ASP.NET Ajax

ASP.NET Ajax состоит из библиотек скриптов и серверных компонентов. Здесь мы рассмотрим их более детально.

Архитектура клиента

Это Microsoft Ajax Library — библиотека клиентских скриптов, состоящая из файлов с кодом на языке JavaScript, позволяющим вам разрабатывать надежные модульные приложения. Всю архитектуру можно разложить по следующим слоям.

- *Компоненты* обеспечивают поддержку функционирования контролов и невизуальных компонентов в браузере без использования обратных посылок.
- *Слой совместимости с браузером* обеспечивает поддержку взаимодействия с большинством браузеров.
- *Слой сетевого взаимодействия* работает с асинхронными запросами и обеспечивает взаимодействие со службами и приложениями, а также сериализацию (XML или JSON).
- *Службы ядра*. К ним относится базовая библиотека классов JavaScript и объектно-ориентированные расширения, которые делают возможным использование классов, наследования, сериализации, обработки событий.
- *Отладка и обработка ошибок*. Это базовый сервис, он включает класс **Sys.Debug**, обеспечивающий нас методами для отображения объектов в удобочитаемом виде в конце Web-страницы. Также он отвечает за показ трассировочных сообщений, разрешает использование макроса ASSERT и точек останова.
- *Глобализация*. Архитектура и сервера, и клиента Ajax обеспечивает модель для локализации глобализации клиентских скриптов. Это позволяет использовать один код для поддержки пользовательского интерфейса на разных языках и культурах.

Архитектура сервера

Архитектура сервера состоит из набора серверных контролов и компонентов, которые можно использовать для создания

интерфейса пользователя и реализации функциональности. ASP.NET Ajax включает в себя следующие компоненты.

- *Поддержка скриптов* реализуется через посылку скриптов с сервера на сторону клиента. В зависимости от того, какие возможности Ajax вы сделали доступными браузеру, посылаются разные скрипты. Также вы можете создать заказной клиентский скрипт. Есть поддержка режимов Debug и Release, есть возможность частично-страничной отправки в асинхронном режиме.
- *Локализация* основана на модели локализации ASP.NET и обеспечивает дополнительную поддержку локализованных файлов .js.
- *Web-службы*. ASP.NET Ajax позволяет клиентскому скрипту обращаться к ASP.NET Web-службам и службам WCF (Windows Communication Foundation).
- *Службы приложений*. ASP.NET Ajax позволяет вам использовать службы приложений, такие как формы аутентификации и профили без обратных посылок. Они встроены в ASP.NET как Web-службы.
- *Специальные серверные контролы*. Эти элементы управления имеют и серверный, и клиентский код для обеспечения развитого поведения. Наиболее часто используются такие контролы, как **ScriptManager**, **UpdatePanel**, **UpdateProgress** и **Timer**.

Раздел 2. Добавление возможностей Ajax стандартным контролам ASP.NET

Как разработчик вы, вероятнее всего, бóльшую часть своего времени посвятите тем или иным аспектам серверной стороны ASP.NET Ajax. Эта технология обеспечивает вас серверными контролами и функциональностью для того, чтобы вы могли привнести Ajax-черты в свое Web-приложение.

Оболочка серверной стороны управляет не только клиентскими запросам, она также управляет взаимодействием и обменом объектов JavaScript и объектов Microsoft .NET на клиентской и серверной стороне. Возможности Ajax

для ASP.NET лежат в основе ASP.NET Framework, WCF и Web-служб на базе ASP.NET.

Основные черты Ajax ASP.NET

Для добавления Ajax-функциональности на отдельной вкладке панели инструментов Visual Studio имеется набор специальных элементов управления. Вы можете использовать такие контролы, как **UpdatePanel**, **UpdateProgress** и **ScriptManager,** для того, чтобы придать вашим Web-формам возможности Ajax; при этом вам не придется писать ни строчки на JavaScript.

Например, контрол **UpdatePanel** даст возможность обновлять только часть страницы вместо загрузки ее целиком. Контрол **ScriptManager** позволит управлять историей браузера через его ссылку «назад».

Контролы серверной стороны являются стандартной частью ASP.NET и включены в Visual Studio. Кроме них, есть обширный набор элементов управления Ajax Control Toolkit. Их можно добавить на панель инструментов среды разработки и затем использовать так же, как и стандартные контролы.

Также вы можете расширить возможности своих Web-форм путем добавления им функциональности клиентской стороны. Клиентские скрипты можно использовать и при создании интерфейса пользователя, и для осуществления асинхронных запросов к Web-серверу, пока страница отображается в браузере.

Встроенные серверные контролы ASP.NET Ajax

Рассмотрим назначение встроенных серверных элементов управления.

- *ScriptManager.* Используется для управления клиентскими скриптами на Web-формах. По умолчанию он регистрирует скрипт для библиотеки Ajax вместе со страницей. Это позволяет скриптовому коду использовать расширения системы типов и поддерживать такие возможности, как частично-страничная отправка и обращения к Web-службам.

  Вы обязаны добавить **ScriptManager** на свою Web-страницу до того, как будете использовать любые другие контролы Ajax.

Если у вас есть контрол **ScriptManager** на мастер-странице, нужно будет добавить элемент управления **ScriptManagerProxy** на те контент-страницы, которые используют контролы Ajax.

Контрол **ScriptManager** также позволяет загружать код библиотеки Microsoft Ajax Library из распределенного по сети хранилища Microsoft CDN.

- *UpdatePanel.* Именно этот элемент управления позволяет выполнять частично-страничную отправку. Web-форма, на которой есть **ScriptManager** и один или несколько **UpdatePanel**, может ее осуществлять автоматически без специально написанного клиентского скрипта. Вы просто перетаскиваете **UpdatePanel** на форму, а уже на нее потом помещаете прочие серверные контролы.
- *UpdateProgress.* Этот элемент управления обеспечивает нас информацией о состоянии при частично-страничной отправке. Вы можете по желанию изменить расположение этого контрола по умолчанию и его содержание.

  Он может работать и с целой страницей и конкретным контролом **UpdatePanel**. Если обновление страницы происходит очень быстро, его можно настроить так, чтобы предотвратить мелькание. Для этого достаточно указать задержку до момента показа этого контрола. Можно также указать, сколько времени его показывать после завершения обновления страницы.
- *Timer.* Этот контрол осуществляет обратные посылки с заданным интервалом времени. Его можно использовать в связке с **UpdatePanel** для периодического обновления страницы как частичного, так и полного.

Использование элемента управления ScriptManager

Для частично-страничного обновления используется связка **ScriptManager** и **UpdatePanel**.

Свойство EnablePartialRendering контрола **ScriptManager** определяет, участвует ли страница в частично-страничном обновлении. По умолчанию его значение равно true, то есть этот режим включен.

Использование расширения системы типов

Microsoft Ajax Library добавляет к JavaScript расширения, которые обеспечивают использование пространств имен, наследования, перечислений, рефлексии и вспомогательных функций для строк и массивов. Тем самым они поднимают функциональность клиентского скрипта почти до уровня языков .NET Framework. Добавляя на форму **ScriptManager**, вы автоматически включаете эти расширения.

Регистрация заказного скрипта

ScriptManager управляет ресурсами, созданными для контролов, участвующих в частично-страничной отправке. Эти ресурсы включают в себя скрипты, стили, скрытые поля и массивы. Коллекция скриптов контрола **ScriptManager** содержит объект **ScriptReference** для каждого скрипта, который доступен браузеру. Указать скрипты можно и декларативно, и программно.

Контрол **ScriptManager** предоставляет вам методы для регистрации, которые можно использовать для программного управления клиентским скриптом и скрытыми полями. Если вы регистрируете скрипт или скрытые поля, которые поддерживают частично-страничное обновление, вы обязательно должны использовать именно эти методы контрола **ScriptManager**. Для регистрации скриптов, которые не нужны для частично-страничного обновления, можно использовать методы класса **ClientScriptManager**.

Регистрация Web-служб

Если из кода клиентской стороны вы хотите вызвать Web-службу, ее требуется зарегистрировать через добавление в коллекцию Services контрола **ScriptManager**. Для каждого объекта **ServiceReference** в этой коллекции оболочка ASP.NET Ajax генерирует клиентский proxy-объект. Классы этих объектов и их строго типизированные члены упрощают использование Web-служб из клиентского скрипта. Добавить в коллекцию Services объекты **ServiceReference** можно и программно во время исполнения.

Использование служб аутентификации, профилей и ролей из клиентского скрипта

ASP.NET Ajax содержит для этого специальные proxy-классы. Если вам нужны заказная служба аутентификации, ее можно зарегистрировать через посредство **ScriptManager**.

Класс ScriptManagerProxy

На странице прямо или опосредованно (через вложенные компоненты, контролы, контент-страницы или вложенные мастер-страницы) может присутствовать только один экземпляр класса **ScriptManager**. Если страница уже содержит объект **ScriptManager**, а вложенному компоненту требуются возможности, предоставляемые этим классом, на него можно поместить объект класса **ScriptManagerProxy**, который сделает возможным использование скриптов и служб, специфичных для этого вложенного компонента.

Использование элемента управления UpdatePanel

Контрол **UpdatePanel** используется для указания области, которая может быть обновлена вместо обновления страницы целиком. Этим на стороне сервера занимается экземпляр класса **ScriptManager**, а на стороне клиента класс **PageRequestManager**.

Если вы делаете возможным частично-страничную отправку, контролы, помещенные внутрь **UpdatePanel**, при выполнении обновления могут асинхронно взаимодействовать с сервером. В ответ на такой запрос сервер посылает HTML-разметку только обновляемых элементов, а в браузере клиентский класс **PageRequestManager** выполняет необходимые DOM-манипуляции для замены существующей разметки.

Как было сказано ранее, такая возможность по умолчанию включена (свойство EnablePartialRendering контрола **ScriptManager** установлено в значение true).

Разберем пример разметки страницы. На ней присутствует контрол **ScriptManager**, также есть панель **UpdatePanel**, а на ней контрол **Button**, который обновляется при щелчке на нем. По умолчанию свойство панели ChildrenAsTriggers установлено в значение true, поэтому кнопка работает в режиме асинхронной обратной посылки.

```
<asp:ScriptManager ID="MainScriptManager"
                   runat="server" />
<asp:UpdatePanel ID="MainUpdatePanel"
                 UpdateMode="Conditional" runat="server">
    <ContentTemplate>
        <fieldset>
            <legend>UpdatePanel content</legend>
            <!-- Остальное содержимое панели -->
            <%=DateTime.Now.ToString() %>
            <br />
            <asp:Button ID="RefreshButton"
                 Text="Refresh Panel" runat="server" />
        </fieldset>
    </ContentTemplate>
</asp:UpdatePanel>
```

Обратите внимание, что здесь свойство панели UpdateMode установлено в значение Conditional. Это означает, что ее содержимое обновляется, только если контрол внутри панели совершает обратную посылку.

Замечание. Если это свойство установить в значение Always, то содержимое панели будет обновляться при посылке, вызванной любым элеменом страницы, а не только элементом внутри **UpdatePanel**.

Как наполнить содержимым UpdatePanel

Это можно сделать как декларативно, так и в режиме дизайнера через свойство ContentTemplate (появляется в разметке при добавлении контролов на панель). Для программного заполнения можно использовать свойство ContentTemplateContainer. В нашем коде элементы fieldset и legend — это пример использования обычных HTML-элементов, которые наряду с серверными контролами могут быть использованы внутри **UpdatePanel**.

При первом отображении страницы, содержащей одну или несколько **UpdatePanel**, все их содержимое посылается браузеру. Но в дальнейшем эта информация может посылаться частями, в зависимости от установок самой панели, размещенных на ней контролов и исполняемого кода.

Использование триггеров на панели обновления

По умолчанию любой postback-контрол, расположенный на панели **UpdatePanel,** вызывает асинхронную обратную

посылку и обновление содержимого панели. Но другие контролы страницы также можно сконфигурировать так, чтобы они вызывали обновление. Это делается через объявление триггеров для контрола **UpdatePanel**. Триггер связывает контрол и событие, которое приводит к обновлению страницы.

Следующий пример показывает, как в разметке прописывается триггер для **UpdatePanel**, здесь это кнопка вне панели:

```
<asp:ScriptManager ID="MainScriptManager"
                   runat="server" />
<asp:Button ID="RefreshButton" Text="Refresh Panel"
            runat="server" />
<asp:UpdatePanel ID="MainUpdatePanel"
                 UpdateMode="Conditional" runat="server">
    <Triggers>
        <asp:AsyncPostBackTrigger
                          ControlID="RefreshButton" />
    </Triggers>
    <ContentTemplate>
        <fieldset>
            <legend>UpdatePanel content</legend>
            <%=DateTime.Now.ToString() %>
        </fieldset>
    </ContentTemplate>
</asp:UpdatePanel>
```

При редактировании страницы в Visual Studio вы можете создавать и настраивать триггеры с использованием диалогового окна UpdatePanelTrigger Collection Editor. Оно доступно через окно Properties.

Задание конкретного события для триггера не является обязательным. Если этого не сделать, то будет использоваться событие по умолчанию (для кнопок это Click).

Как происходит обновление контролов в UpdatePanel

Ниже мы перечисляем установки свойств, влияющих на процесс обновления.

- Если свойство UpdateMode равно Always, то панель обновляется при любой обратной посылке от любого контрола на странице.

- Если свойство UpdateMode равно Conditional, панель обновляется при выполнении любого из следующих условий:
    - postback вызван триггером для этой панели;
    - был явно вызван метод Update для этой панели;
    - панель вложена в другую панель, и родительская панель обновляется;
    - свойство ChildrenAsTriggers равно true, в этом случае любой элемент панели срабатывает как триггер, дочерние контролы вложенной панели **UpdatePanel** не вызывают обновления для родительской панели, если только они явно не определены для нее как триггеры.

Замечание. Если ChildrenAsTriggers равно false, а UpdateMode равно Always, будет выброшено исключение, поскольку такое сочетание является недопустимым. Свойство ChildrenAsTriggers можно задавать, только если UpdateMode равно Conditional.

Использование элемента управления UpdateProgress

Контрол **UpdateProgress** отрисовывает элемент div, который виден или спрятан в зависимости от того, вызвал ли связанный с **UpdatePanel** контрол асинхронную обратную посылку. При первом показе и при синхронной посылке он не отображается.

О связывании с UpdatePanel

Связь с **UpdatePanel** осуществляется через установку свойства AssociatedUpdatePanelID. Как только для этой панели срабатывает событие на обратную посылку, контрол **UpdateProgress** становится видимым. Если вы не связали его ни с какой панелью, он будет отображать ход любой асинхронной посылки.

Если у панели свойство ChildrenAsTriggers установлено в false и посылка инициирована изнутри панели, все связанные с ней контролы **UpdateProgress** будут отображаться.

О размещении на странице

Если свойство DynamicLayout установлено в значение true, то изначально контрол **UpdateProgress** не занимает места

на странице. Когда его содержимое нужно будет отобразить, страница динамически изменится. В этом случае контрол отображается как элемент div со свойством display, равным none.

Если свойство DynamicLayout установлено в значение false, контрол **UpdateProgress** занимает место на странице, даже если он пока не виден. У соответствующего элемента div свойство display устанавливается в значение block, а свойство visibility изначально равно hidden.

Разместить контрол **UpdateProgress** можно как на самой панели **UpdatePanel**, так и вне ее. Отображаться он будет только в процессе обратной посылки, даже если размещен в другой панели.

Если наш экземпляр UpdatePanel вложен в другую панель, процесс обратной посылки, инициированный изнутри дочерней панели, приводит к отображению всех контролов **UpdateProgress**, с нею связанных. Но вместе с этим будут показаны и контролы **UpdateProgress**, связанные с родительской панелью. Если обратная посылка вызвана элементом на родительской панели, то отображаться будут только контролы **UpdateProgress**, связанные с нею.

Можно также задать временную задержку для отображения **UpdateProgress** после начала посылки. Для этого есть свойство DisplayAfter, задающее этот промежуток времени в миллисекундах.

О программном управлении частично-страничной посылкой на стороне клиента

При использовании частично-страничного обновления на вашей странице доступен экземпляр класса **PageRequestManager**, обеспечивающий полное управление этим процессом. Получить доступ к этому объекту можно через вызов метода getInstance данного класса, а для полученного экземпляра можно уже вызывать остальные методы класса **PageRequestManager**, например abortPostBack или Dispose.

Замечание. Самостоятельно создавать этот экземпляр нельзя, только через getInstance.

Класс **PageRequestManager** является членом пространства имен **Sys.WebForms** из библиотеки Microsoft Ajax Library. Он относится к инфраструктуре клиента Ajax, которая обеспечивает

автоматическую привязку событий (аналогично событиям для серверных контролов).

Ниже мы приводим краткую справку о моментах, когда возникают события класса **PageRequestManager,** и действиях, которые можно было бы выполнить в обработчике.

- *initializeRequest.* До начала обработки запроса на асинхронную обратную посылку. Можно ее отменить.
- *beginRequest.* До начала асинхронной обратной посылки. Можно предоставить пользователю информацию о состоянии.
- *pageLoading.* Запрошенная информация от сервера уже получена, но отображаемая страница еще не изменена. Можно внести какие-нибудь изменения.
- *pageLoaded.* Отображаемая страница уже изменена. Можно внести какие-нибудь изменения.
- *endRequest.* Управление передано браузеру. Можно предоставить пользователю информацию о состоянии.

Приведем кусочек кода, иллюстрирующий получение экземпляра класса **PageRequestManager**, а также формат обработчика события на примере события beginRequest:

```
<script type="text/JavaScript" language="JavaScript">
    Sys.WebForms.PageRequestManager.getInstance().
                        add_beginRequest (ReqHandler);
    function ReqHandler(sender, args)
    {
        ...
    }
</script>
```

Раздел 3. Элементы управления Ajax из пакета Ajax Control Toolkit

При установке среды разработки Visual Studio вы получаете только несколько элементов управления, позволяющих создать Web-приложение с возможностями Ajax. Однако сообщество разработчиков предлагает постоянно развивающийся обширный набор серверных контролов с Ajax-возможностями. Он называется Ajax Control Toolkit и содержит множество как самодос-

таточных контролов, так и расширителей, которые вы можете использовать в своих приложениях.

Обзор пакета Ajax Control Toolkit

Отметим, что Microsoft развитием и поддержкой этого пакета не занимается, однако он легко встраивается в среду разработки Visual Studio.

Библиотека содержит коллекцию компонентов, которые вы можете встраивать в свои Web-приложения. Также вы можете использовать инфраструктуру пакета, чтобы создавать свои Ajax-контролы и расширители.

Замечание. Расширитель (экстендер) — это не самостоятельный элемент управления, он предназначен для того, чтобы добавлять определенную функциональность к уже существующему контролу.

Для установки пакета Ajax Control Toolkit вам нужно скопировать из сети его архив и распаковать. Вы получите следующие папки с его компонентами.

- *Папка Scripts*. Она содержит файлы с кодом на JavaScript.
- *Папка AspNetAjaxBetaSamples*. Это примеры кода для демонстрации возможностей пакета, оформленные как решения Visual Studio.
- *Папка Web Forms*. Она содержит две сборки, которые требуются для использования контролов из данного пакета в качестве серверных элементов управления в Web-приложениях, а именно: System.Web.Ajax.dll и AjaxControlToolkit.dll.

Обзор контролов Ajax Control Toolkit

Ajax Control Toolkit содержит несколько десятков контролов. Их можно попробовать в деле, собрав и запустив на выполнение проект из решения BetaSamples.sln.

Элементы управления легко встраиваются в панель инструментов (Toolbox) Visual Studio. После этого вы работаете с ними обычным образом. Описание всех предлагаемых контролов было бы слишком объемным, поэтому мы перечислим только несколько из них, чтобы можно было сформировать представление о возможностях пакета.

- *Accordion*. Позволяет управлять набором панелей, отображая в каждый момент только одну из них.
- *AutoCompleteExtender*. Расширитель — прицепляется к полю ввода. Выглядит как всплывающая панель с предлагаемыми вариантами заполнения поля.
- *CalendarExtender*. Расширитель — прицепляется к полю ввода, позволяет сделать ввод даты более удобным. Формат даты можно настроить, исходя из своих предпочтений.
- *ComboBox*. Название говорит само за себя. Похож на AutoCompleteExtender и DropDownList. Имеет набор опций для настройки.
- *ConfirmButtonExtender*. Перехватывает клики на кнопке и выдает сообщение пользователю, где предлагает подтвердить действие.
- *DragPanelExtender*. Позволяет добавить к панели возможности drag-and-drop.

Мы ограничимся этим кратким перечнем. Для получения более полного представления о возможностях пакета полезнее будет загрузить его из сети Интернет и установить на свой компьютер.

Лекция 12. Использование возможностей служб WCF

Создание приложений, управляемых данными (data-driven), ускоряет и упрощает разработку. Они хорошо подходят для совместной работы в интрасети. Однако эта техника не всегда годится для Интернета, главным образом, из соображений безопасности.

Обеспечение корректного функционирования кода в большой сети — это не то же самое, что обеспечить функциональность в пределах класса или компонента. Потенциальные пользователи в силу очевидных причин могут не иметь вашего кода или не иметь прав на его использование. Нам же нужно обеспечить нормальное его функционирование без предоставления пользователям (партнерам, заказчикам и т.д.) полного доступа к коду. При этом, если вам требуется еще и безопасное соединение, вы можете, конечно, создать front-end заслон, который предоставляет функциональность и данные в недоверительной манере, как это делается при предоставлении всем доступа с использованием протоколов HTTP или HTTPS.

Однако для многих компаний даже с такими дополнениями вариант использования data-driven Web-страниц остается неприемлемым, поскольку не удовлетворяет их бизнес-потребностям. Более удачное решение — использование программ с логикой и функциональностью, которая напрямую связывает организации, приложения и данные.

Именно такой вариант обеспечивают службы WCF (Windows Communication Foundation). Используя их, можно существенно расширить функциональность, предлагаемую пользователям.

Далее мы узнаем, как использовать службы WCF напрямую из Web-приложения, а также как работать со службами данных WCF, чтобы сделать наши данные доступными через глобальную сеть.

Раздел 1. Введение в службы WCF

При создании развитого Web-сайта нам нужно решить, как мы обеспечим взаимодействие образующих его приложений. Зачастую для этого мы просто собираем несколько приложений в рамках одного решения.

Проблема здесь заключается в том, что собираемые приложения могут быть разработаны для разных платформ, а платформы — для разных операционных систем. Кроме того, приложения еще могут быть написаны на разных языках.

Идея SOA (service-oriented architecture) заключается в осуществлении взаимодействия посредством служб, и в настоящее время это наиболее распространенный подход к решению сформулированной проблемы.

Разумеется, идея должна быть воплощена в соответствующие инструменты и технологии. Службы WCF специально для этого и разработаны.

Что такое Windows Communication Foundation?

Глобальная доступность Web-служб с использованием стандартных протоколов взаимодействия приложений существенно изменила сам процесс разработки программного обеспечения. Например, для обеспечения безопасных и надежных распределенных транзакций зачастую применяются именно Web-службы.

Все это отразилось и на используемых разработчиками инструментах и технологиях. Поэтому Microsoft и разработала WCF, специально предназначенную для распределенных вычислений с ориентацией на поддержку со стороны служб.

WCF является частью .NET Framework, обеспечивающей программную модель для быстрой разработки приложений, ориентированных на службы, и их взаимодействие через сеть. Она в некотором роде пришла на смену более ранним технологиям XML Web-служб и ремоутинга (Remoting).

Модель WCF упрощает разработку взаимосвязанных приложений и поддерживает множество стилей распределенного программирования. Она имеет многоуровневую архитектуру. В ее основании канальная архитектура WCF предоставляет нам примитивы для асинхронной нетипизированной передачи сообщений. Лежащие выше уровни обеспечивают безопасный, надежный обмен данными с широким выбором транспортных протоколов и кодировки.

Разработанная ранее типизированная программная модель характеризовалась прямым отображением концепций XML Web-служб на концепции CLR и соответствующие языки программирования.

Используя WCF, вы можете выстроить безопасное и надежное кроссплатформенное решение, обеспечивающее асинхронную передачу данных от одной точки обслуживания (endpoint) к другой. Эти точки могут быть, например, частью службы, размещенной в IIS (Internet Information Services) и функционирующей постоянно, или частью вашего приложения. Клиентская часть запрашивает данные, и сообщение в текстовом (обычно XML) или бинарном виде через поток направляется в точку обслуживания.

О взаимодействии с приложениями, использующими другие технологии

При разработке WCF возможность взаимодействия с другими (не WCF) приложениями была заложена изначально, даже если те разрабатывались для других платформ или основывались на более ранних технологиях Microsoft.

Основу этого механизма обеспечивают XML Web-службы и формат данных SOAP (Simple Object Access Protocol). Вот с какими типами приложений возможно такое взаимодействие:

- WCF-приложения, запущенные в другом процессе на данном компьютере под управлением Windows;
- WCF-приложения, запущенные на другом компьютере под управлением Windows;
- Приложения, основанные на других технологиях, таких как J2EE (Java 2 Enterprise Edition), поддерживающих стандартные XML Web-службы.

Благодаря этому WCF не препятствует использованию существующих технологий и не требует внесения изменений в ранее разработанные приложения с целью использования WCF.

Что такое служба WCF?

Служба WCF — это приложение, оформленное как служба, которая может взаимодействовать с другими приложениями, используя стандарты Интернет, такие как HTTP, XML, протокол TCP (Transmission Control Protocol) и другие. В каком-то смысле такая служба — это «черный ящик», который расширяет функциональность приложений-клиентов.

Рассмотрим основные характеристики служб WCF.

- Они могут взаимодействовать с другими приложениями по сети (в том числе через Интернет), не требуя наличия постоянного соединения.
- Как любая служба, они не имеют пользовательского интерфейса, вместо этого они предлагают так называемый контракт — набор интерфейсов, которые описывают, что служба может делать.
- Использовать ее локально или через глобальную сеть может одно или одновременно несколько приложений.

Использование служб WCF дает заказчикам и разработчикам следующие преимущества.

- *Независимость от языка программирования*. Годится любой язык для программирования в .NET Framework.
- *Независимость от протоколов*. WCF-службы не используют протоколов, специфичных для некоторых объектных моделей (типа DCOM). Используются только стандартные Web-протоколы и форматы данных: HTTP, XML, SOAP. Любой сервер, поддерживающий эти протоколы, может иметь доступ к WCF-службе или размещать ее у себя.
- *Независимость от платформы*. Использование стандартных протоколов позволяет самым разным системам работать вместе. Серверы, которые поддерживают Web-формы, также поддерживают и WCF-службы.
- *Обмен сообщениями основан на одном или нескольких шаблонах*. Чаще всего используется шаблон запрос/ответ. Есть и другие варианты, например однонаправленная посылка сообщения (ответа не ждем) или более сложный вариант — установка постоянного двунаправленного соединения и обмен данными по нему.
- *WCF поддерживает публикацию метаданных службы* в соответствии с промышленными стандартами, такими как WSDL (Web Services Description Language), XML Schema и WS-Policy. Они позволяют автоматически сгенерировать необходимый код и настроить конфигурацию клиента для использования службы. Опубликовать метаданные можно по протоколам HTTP и HTTPS или

с использованием специального стандарта Web Service Metadata Exchange.

- *Использование контрактов данных*. Поскольку WCF взаимодействует с .NET Framework, он включает в себя дружественные способы для поддержки контрактов, которые вы желали бы использовать. Один из таких контрактов — контракт данных. Благодаря ему у вас есть простой способ доступа к данным путем создания классов на привычном языке программирования. Эти классы представляют сущности ваших данных и имеют необходимые свойства. WCF предоставляет для этого все необходимые средства.

Взаимодействие через WCF-службы

Все взаимодействие через WCF-службы происходит между конечными точками (endpoint) службы. Конечная точка — это то, что обеспечивает клиенту доступ к функциональности, предлагаемой WCF-службой. Она содержит четыре составляющих, реализованных как свойства:

- Address однозначно идентифицирует конечную точку и сообщает потенциальным пользователям, где размещается служба; в объектной модели WCF она представлена классом **EndpointAddress**, который имеет следующие свойства:
  - Uri представляет адрес службы;
  - Identity представляет сущность службы и содержит коллекцию дополнительных (необязательных) заголовков сообщений.
- Binding — привязка, она определяет, как происходит взаимодействие с конечной точкой, включает в себя следующую информацию:
  - какой используется транспортный протокол;
  - какой используется поток, бинарный или текстовый;
  - необходимые требования безопасности, например использование SSL (протокол защищенных сокетов).

- Contract — предоставляемая функциональность, контракт определяет:
  - какие операции могут использоваться клиентом;
  - форму сообщения;
  - тип входных параметров или данных, требующихся для выполнения операции;
  - какого типа обработки или ответного сообщения может ожидать клиент.
- Behaviors. Определяет локальное поведение конечной точки службы. Пример поведения: свойство ListenUri — набор адресов, которые мы будем прослушивать.

Области использования WCF-служб

Службы WCF позволяют разделять логику и возможности программирования с многочисленными Web-приложениями и приложениями Windows, а также с приложениями, работающими на других платформах. Можно представлять себе службу WCF как компонент, который может предоставлять свои методы через сеть.

Такого рода компоненты весьма востребованы, приведем некоторые примеры областей, где их широко используют:

- службы аутентификации;
- информация о погоде;
- информация об обменных курсах валют;
- резервирование билетов;
- игра на бирже;
- сервис, предоставляемый вашими партнерами;
- заголовки новостей;
- отслеживание заказов.

Преимуществом использования служб является также то, что программы могут быть написаны на разных языках и для разных платформ, при этом хорошо взаимодействуя.

Вы можете предоставить кому-либо определенную функциональность, но не давать прямого доступа к своему коду или компоненту. Также вы можете предоставлять своей клиентуре функциональность третьих лиц, как свою.

Раздел 2. Вызов методов служб WCF из Web-формы

Раздел посвящен вопросам программного доступа к сервису, предоставляемому WCF-службой.

Использование службы Windows Communication Foundation

WCF-служба, разумеется, сначала должна быть где-то размещена, рассмотрим возможные варианты:

- консольное приложение;
- Windows-приложение с графическим интерфейсом;
- IIS;
- служба Windows;
- служба WAS (Windows Process Activation Service).

Первые два варианта еще называют self-hosting, поскольку, для того чтобы WCF-служба стала доступной, их просто нужно запустить на выполнение.

Вариант «служба Windows» означает, что WCF-служба запускается внутри другой службы Windows.

Вариант с использованием WAS позволяет размещать WCF-службу и поддерживать любой транспорт внутри IIS-модели. При этом WAS создает рабочие процессы и обеспечивает конфигурирование. Если IIS делает Web-приложения или Web-службы доступными через протокол HTTP, то WAS делает их доступными и через другие протоколы (HTTP, TCP) и даже через именованные каналы.

Обнаружение Web-служб

Web-службы публикуются и проходят регистрацию UDDI (Universal Description, Discovery, and Integration). Вы можете найти Web-службу прямо из среды разработки Visual Studio и сделать ее частью своего решения. Если вы работаете с локальной службой в IIS, ее адрес выглядит примерно так: *http://localhost/CustomerWCF/Customers.svc*. Обратите внимание, что расширение .svc нужно указывать обязательно.

Использование Web-служб

Использовать Web-службу можно по-разному. Если вы ее используете на стороне клиента, с помощью специальных

инструментов, вы можете сгенерировать код proxy-классов. WCF обеспечивает и стандарты, и инструменты для этого (утилита SvcUtil.exe). Эта утилита генерирует необходимый код на основании полученных метаданных. В Visual Studio достаточно просто добавить в проекте ссылку на службу и код будет сгенерирован автоматически.

Как добавить ссылку на службу в проект Visual Studio

Перечислим шаги, которые требуется выполнить в среде Visual Studio для добавления ссылки на службу в свой проект.

1. В диалоговом окне *Add Service Reference* выбираем вариант *Discover*, далее вводим URL службы в окне *Address* и нажимаем кнопку *Go*. Появится список доступных служб. Каждая может содержать несколько контрактов и/или конечных точек.
2. Выбираем из списка контракт, в окне *Operations* нам выводят набор операций, доступных по данному контракту. Щелкаем по кнопке *OK*.
3. В ответ Visual Studio загружает описание службы на наш компьютер и генерирует класс proxy для выбранной службы и контракта. Этот класс содержит методы для вызова каждого из методов службы, прописанного в контракте. Сам класс содержится в файле с фоновым кодом для локального файла .wsdl.

Обзор файлов с метаданными ссылки на службу

После завершения процесса добавления ссылки на WCF-службу в папке App_WebReferences появляется несколько файлов с метаданными. Имена большинства из них производятся от имени службы, поэтому при перечислении мы будем указывать только расширение файла.

- *Reference.svcmap*. Это главный файл в формате XML, содержащий ссылки на файлы метаданных и операции для генерации кода. В нем элементы MetadataFile указывают на файлы с расширениями .wsdl, .xsd и .disco, а элемент ExtensionFile на файлы конфигурации.

- *.disco.* Файл содержит элемент contractRef, который ссылается на другие документы, включая WSDL и документацию на сервере. Этот файл не используется ни для конфигурирования, ни для генерации proxy.
- *.wsdl.* Файл содержит WSDL-разметку для службы.
- *.xsd.* Файлы в формате XML с описанием используемых схем.
- *configuration.svcinfo* и *configuration91.svcinfo.* Это информация, добавляемая в файл Web.config.

Раздел 3. Службы данных WCF

Развитие технологий, появление Ajax и Silverlight радикально изменили представление о том, в каком виде, с какой функциональностью и какие вообще данные должен получать Web-браузер клиента.

В традиционных Web-приложениях серверная сторона создает (рендерит) HTML-контент, который содержит шрифты, цвета, описание расположения данных и собственно данные, а также код клиентской стороны, скажем на JavaScript. Один из ключевых моментов новых архитектурных решений заключается в том, что данные и их представление не упаковываются в один контейнер и не доставляются по одному каналу.

Страницы Web-сайтов на основе Ajax имеют и представление, и функциональность (код на JavaScript), причем данные доставляются отдельно с использованием XML и HTTP. Приложения Silverlight вообще убирают с серверной стороны процесс рендеринга, который смешивает данные и код. Код для управления различными аспектами представления проходит предкомпиляцию и размещается в файле на Web-сервере. По достижении Web-браузера клиента код обращается к Web-серверу за получением реальных данных для отображения в пользовательском интерфейсе.

Сейчас в Интернете нередко используется новый тип приложений, которые называют гибридными приложениями. Он использует понятие mashups — внешних интерфейсов, которые агрегируют и объединяют данные, доступные в сети в чистом виде. В настоящее время большинство из них

представлены в форме каналов RSS (Really Simple Syndication) или Atom. Эта архитектура порождает интерес к услугам передачи данных, то есть сервисам, которые приложения могут использовать для поиска и обработки данных в сети независимо от технологии представления, используемой в пользовательском интерфейсе, или от того, на каком сервере размещены данные.

В рамках данного раздела мы познакомимся с тем, какие возможности в этом плане нам предлагают службы данных WCF.

Обзор служб данных WCF

Оболочка служб данных WCF (WCF Data Services) состоит из комбинации шаблонов и библиотек для создания и использования таких служб.

Службы WCF используют URI (Uniform Resource Identifier) для адресации порций данных и простые, хорошо известные форматы их представления: JSON или ATOM. В результате служба данных отображается как коллекция ресурсов, к которой можно обращаться с помощью URI и с которой можно взаимодействовать, используя стандартные конструкции HTTP, такие как GET, POST, PUT или DELETE.

WCF Data Services для представления данных обычно предполагает использование модели EDM (Entity Data Model), которая организует их как экземпляры сущностей и набор связей между ними.

Для реляционных данных WCF Data Services использует модель EDM и оболочку ADO.NET Entity Framework. Для доступа к нереляционным данным или при использовании иных технологий доступа (скажем, LINQ to SQL) предусмотрен механизм, который позволяет моделировать любой источник данных как объект, который можно рассматривать в качестве службы данных.

Службы WCF используют простые форматы для представления данных и поддерживают более одного формата, чтобы удовлетворить как можно больше клиентов. Для проведения аутентификации эти службы используют обычные схемы.

Представление данных в WCF Data Services не зависит от типа источника. Служба принимает HTTP-запрос к ресурсу, идентифицируемому через URI, десериализует его и передает поставщику службы для выполнения на источнике.

Такое разделение протоколов доступа позволяет службам представлять данные на том уровне абстракции, который был предложен поставщиком и который может отличаться от схемы собственно хранилища.

Возможности работы с реляционными базами данных расширяются также и тем фактом, что WCF Data Services могут использовать хранимые процедуры.

Создание службы данных WCF

Серверная оболочка WCF Data Services состоит из двух половин: верхней и нижней. Верхняя — это собственно процесс исполнения, она постоянна и реализует трансляцию URI, форматы передачи Atom/JSON и протоколы взаимодействия. Нижняя часть — это уровень доступа к данным, она подключаемая. Сообщение между слоями происходит в терминах интерфейса **IQueryable** и основано на соглашениях по привязке CLR к шаблонам WCF Data Services.

Выбор источника данных

Первый шаг при создании службы данных WCF — определить источник данных, который будет представлен как набор конечных точек. Вам требуется выбрать или создать слой доступа к данным.

Для реляционных данных из базы под управлением SQL Server или иных баз данных WCF Data Services позволяет вам просто взять концептуальную модель, созданную с использованием ADO.NET Entity Framework. А такая технология, как LINQ to SQL, обеспечивает работу с данными из любых источников, в том числе нереляционных.

Последовательность шагов для создания службы данных с использованием ADO.NET Entity Framework

1. Создать в Visual Studio новый Web-проект или открыть существующий.
2. Создать EDM-представление вашей базы данных, используя ADO.NET Entity Framework.

   Для представления 1:1 достаточно в диалоговом окне *Add New Item* выбрать шаблон *ADO.NET Entity Data Model*. Созданный файл получит расширение .edmx.

3. Создать собственно службу.

   В диалоговом окне *Add New Item* выбираем шаблон *WCF Data Service*. Созданный файл получит расширение . svc. В файле с фоновым кодом для него найти подсказку, оформленную в виде комментария и добавить туда имя класса данных, включая пространство имен.

4. Сделать службу доступной.

   По умолчанию служба не выставляет никаких ресурсов из соображений безопасности. Доступ нужно включить явным образом для каждого из ресурсов.

   Для включения чтения/записи в коде модели EDM, связанной со службой, надо найти метод InitializeService, раскомментарить там один из методов (например, метод SetEntitySetAccessRule) и задать в качестве первого параметра имя сущности или звездочку ('*' означает «для всех»).

5. Протестировать службу.

   Запустить Web-приложение или Web-сайт, добавить в строке URL имя службы с расширением .svc, например WebDataService.svc, и нажать ENTER. Вы должны получить данные из источника в формате XML.

Лекция 13. Управление состоянием Web-приложений в ASP.NET

Протокол HTTP по определению является бесстатусным протоколом, то есть он не сохраняет состояние диалога между сервером и клиентом. Каждый запрос обслуживается по приходу, а по завершении его обслуживания все данные, связанные с соединением и запросом, теряются. Никакой информации до следующего запроса не сохраняется, будь это тот же самый клиент или какой-либо другой.

Тем не менее для очень широкого спектра программных решений очень важно иметь возможность хранить такую информацию, и ASP.NET ее нам предоставляет, позволяя сохранять состояние Web-приложения между запросами.

Раздел 1. Управление данными приложения

Соединение, которое устанавливается между клиентом и Web-сервером, принято называть сеансом (session). Сеанс может охватывать несколько страниц, при переходе между которыми нужно сохранять некоторую информацию. Впрочем, информацию может потребоваться сохранять и между запросами одной и той же Web-страницы. Это и называется сохранением состояния сеанса.

Также ASP.NET позволяет сохраненить состояние приложения между сеансами и запросами страниц данного Web-приложения, обеспечивая, таким образом, глобальный охват.

В данном разделе мы узнаем, почему так важно уметь управлять состоянием, как это делать и в чем заключается разница в управлении состоянием на стороне сервера и на стороне клиента.

Зачем нужно управление состоянием?

Как мы уже отметили, ASP.NET использует бесстатусный протокол HTTP, а это означает, что Web-формы даже не могут определить, приходят ли запросы от одного и того же клиента, даже если он, не выходя из браузера, просматривает одну страницу за другой. Более того, на каждый новый запрос отображение Web-страницы генерируется сервером заново (если

нет кэширования), так что информация на странице не существует за пределами ее жизненного цикла.

ASP.NET предлагает механизм управления, который сохраняет на сервере данные между запросами и обратными посылками. Таким образом, мы можем сохранять информацию пользователя в процессе его работы на сайте. В частности, это позволяет ему повторно вводить свои данные, скажем, при заполнении какой-либо сложной формы.

Типы управления состоянием

Вы можете управлять состоянием своего приложения как на стороне сервера, так и на стороне клиента. Выбор в основном определяется природой вашего Web-приложения.

Основная разница между этими двумя вариантами состоит в производительности и степени безопасности. Если для хранения информации используются ресурсы сервера, то тем самым достигается бóльшая безопасность. При управлении на стороне клиента уровень безопасности минимальный, зато не требуется пересылать информацию на сервер и обратно.

Управление на стороне сервера

Управление на стороне сервера обеспечивает нам следующие возможности.

- *Состояние приложения*. Это информация, доступная всем пользователям Web-приложения. Например, вы можете хранить количество посетителей вашего сайта.
- *Состояние сеанса*. Это информация, доступная только пользователю конкретного сеанса. Например, это полномочия данного клиента.
- *Профиль*. Свойства профиля позволяют сохранять данные о конкретном пользователе между его сеансами работы, например, его адрес, телефон и т. п.
- *Кэширование*. Обеспечивает возможность сохранения вычисленных значений или всей страницы, чтобы не выполнять работу повторно.

Управление на стороне клиента

На стороне клиента нам доступны следующие инструменты.

- *ViewState*. Это специальная структура (скрытое бинарное поле), позволяющая автоматически между запросами одной и той же страницы передавать серверу информацию. По умолчанию эта возможность включена.
- *ControlState*. Это свойство, позволяющее сохранять настроечные данные элемента управления при запросах страниц. Такая возможность особенно полезна, когда вы создаете свои заказные контролы, поскольку ControlState, в отличие от ViewState, нельзя отключить.
- *Скрытые поля*. Работают как хранилища информации, специфичной для вашей страницы.
- *Куки (cookies)*. Маленький текстовый файл (до 4 или до 8 килобайт) с плоским незащищенным текстом.
- *Строки запроса*. Они прицепляются к URL при обращении к серверу, пример:

```
http://www.contoso.com/Customers.aspx?name=Baggins
```

На стороне сервера хранить информацию безопаснее, и ее может быть намного больше, что важно для расширяемости вашего приложения. На стороне клиента информации можно хранить меньше, зато все работает быстрее, не обеспечивая, впрочем, серьезного уровня безопасности. При выборе между двумя этими вариантами вы можете принимать во внимание следующие соображения:

- Сколько информации вам нужно хранить?
- Поддерживает ли клиент постоянные файлы cookie или файлы cookie в памяти?
- Хотите ли вы хранить информацию на стороне клиента или на стороне сервера?
- Является ли информация конфиденциальной?
- Какие критерии производительности и пропускной способности вы установили для своего приложения?
- Каковы возможности браузеров и устройств, на которые вы ориентируетесь?
- Нужно вам ли хранить информацию для каждого пользователя?
- Как долго вам нужно хранить информацию?

- Где запущено ваше Web-приложение: на Web-ферме (несколько серверов), в виде нескольких процессов на одном компьютере (Web garden) или это один процесс?

Средства управления состоянием на стороне сервера

Состояние приложения

ASP.NET обеспечивает управление состоянием приложения через использование экземпляра класса **HttpApplicationState**. Это глобальный механизм хранения, доступный из всех страниц Web-приложения. Следовательно, он позволяет сохранять информацию между посылками данных, при переходах между страницами для разных сеансов и разных пользователей.

Состояние приложения хранится в виде коллекции пар "ключ-значение", которая создается при запросе конкретного URL. Вы можете добавить информацию для хранения в этой коллекции даже между запросами страниц. После этого сервер будет управлять ее хранением в памяти. При перезапуске приложения эта информация сбрасывается. Доступна она через экземпляр класса **HttpApplicationState**, а тот, в свою очередь, через свойство Application класса **Page**. Хранятся составные элементы этой информации в виде объектов типа **System.Object**.

Переменные приложения

Это отдельные фрагменты информации, хранящейся в Application state. Самые подходящие для такого хранения данные — это те, которые требуется сохранять между сеансами и которые не очень часто меняются. Рассмотрим пример того, как можно сохранить и прочитать такие данные:

```
// Сохранить переменную приложения
Application["ApplicationName"] =
                                    "The best application";
// Прочитать переменную приложения по имени
string appName = (string)
Application["ApplicationName"];
// Прочитать переменную приложения по индексу
string appName = (string) Application[0];
```

Обратите внимание на преобразование типов при чтении, так как данные хранятся в виде **System.Object**. Перед доступом

к данным разумно было бы проверить их наличие, чтобы избежать исключений из-за использования нулевых ссылок:

```
// Существует ли переменная приложения?
if (Application["ApplicationName"] != null)
```

Чтобы быть уверенным, что ваши переменные приложения были размещены при старте, их лучше добавлять в методе Application_Start (в файле Global.asax), например, так:

```
void Application_Start(object sender, EventArgs e)
{
    Application["ApplicationName"] =
                                    "The best application";
}
```

Замечание. При доступе к переменным приложения класс **HttpApplicationState** осуществляет автоматическую блокировку и разблокировку, используя свойства AllKeys и Count, а также методы Add, Clear, Get, GetKey, Remove, RemoveAt и Set. Однако если вам требуется выполнить несколько действий, то будет более эффективным применить ручную синхронизацию посредством методов Lock и UnLock.

Переменные приложения существуют, пока не произойдет событие Application_End. Это происходит непосредственно перед уничтожением последнего экземпляра приложения.

Соответствующий обработчик размещается в Global.asax, этот файл опциональный, он создается по вашему требованию, если вы хотите работать с переменными приложения и/или сеанса или инициализировать их из конфигурационного файла.

Состояние сеанса

Управление состоянием сеанса обеспечивается через экземпляр класса **HttpSessionState**, который создается для каждого активного сеанса.

Это очень похоже на работу с состоянием приложения, только все ограничивается текущим сеансом связи с браузером. Если с приложением одновременно работают несколько пользователей, у каждого будет свое состояние сеанса. Если вы покинули Web-приложение и вернулись позже, состояние сеанса будет другим.

Состояние сеанса хранится в виде коллекции пар "ключ-значение", информация сохраняется между циклами запроса/отправки или переходами с одной страницы на другую. По умолчанию эти данные хранятся только в памяти Web-сервера, однако существует возможность настроить ASP.NET на автоматическую сериализацию информации о сеансе в базе данных или на специальном сервере. Такая возможность включается в файле Web.config, соответствующая конструкция выглядит примерно так:

```
<sessionState mode="SQLServer"
    sqlConnectionString="Data Source=ServerName;
    Integrated security=true" />
```

или так:

```
<sessionState mode="StateServer"
    stateConnectionString="tcpip=ServerName:42424" />
```

Вопросы настройки этого сервера и запуска соответствующей службы мы здесь не рассматриваем.

Переменные сеанса

Лучшие кандидаты на хранения в таком виде — короткоживущие данные, привязанные к конкретному сеансу. Для обеспечения большей безопасности вы можете задать тайм-аут для сеанса, то есть время жизни таких переменных.

Рассмотрим пример кода для сохранения и чтения информации из переменных сеанса:

```
// Сохранить переменную сеанса
Session["Username"] = "Santa Claus";
// Прочитать переменную сеанса по имени
string username = (string) Session["Username"];
// Прочитать переменную сеанса по индексу
string username = (string) Session[0];
```

Обратите внимание на преобразование типов при чтении, так как данные хранятся в виде **System.Object**. Перед доступом к данным, как и для переменных приложения, разумно было бы проверить их наличие, чтобы избежать исключений из-за использования нулевых ссылок:

```
// Существует ли переменная сеанса?
if (Session["Username"] != null)
```

Для гарантированной инициализации соответствующий код можно поместить в метод Session_Start (в файле Global.asax), например, так:

```
void Session_Start(object sender, EventArgs e)
{
    // Сохранить переменную сеанса
    Session["Username"] = "Santa Claus";
}
```

Идентификация и отслеживание сеанса

Каждое активное Web-приложение идентифицируется строкой из 24 символов, разрешенных для использования в URL. Эта строка прописывается в куки (cookie) на клиентском компьютере. Если у клиента куки запрещены, этот идентификатор добавляется непосредственно в URL. Такую возможность можно включить в файле Web.config:

```
<configuration>
    ...
    <system.web>
        ...
        <sessionState cookieless="UseUri" />
    </system.web>
    ...
</configuration>
```

А так примерно будет выглядеть строка для запроса страницы:

```
http://www.server.com/(S(bjyirwq4i1j42versw2wq53u))/default.aspx
```

Замечание. Если вы используете бескуковые сеансы, вы не можете на странице использовать абсолютные URL для перехода на другую страницу — адрес должен задаваться только относительно текущей страницы. Кроме того, если ваш сайт использует Ajax, в качестве значения атрибута cookieless следует использовать только значение по умолчанию UseCookies, так как включение куки в URL не поддерживается библиотеками клиентских скриптов ASP.NET Ajax.

Тайм-аут сеанса

По умолчанию, если пользователь не сделал очередной запрос в течение 20 минут, сеанс завершается, далее он рассмат-

ривается как новый пользователь. Такое поведение позволяет экономить ресурсы сервера. Настройка этого параметра может быть сделана в файле Web.config:

```
<configuration>
    <system.web>
        <sessionState timeout="30" />
    </system.web>
</configuration>
```

Свойства профиля

Так же, как и переменные сеанса, они позволяют хранить специфичные для пользователя данные, если только они не привязаны к сеансу. Профиль позволяет легко управлять информацией пользователя, не требуя поддержки собственной базы данных. Кроме того, профиль делает возможным доступ к пользовательской информации с использованием строго типизированного интерфейса откуда угодно в пределах приложения. В профиле можно хранить объекты любого типа.

Кэширование

Вы можете увеличить производительность своего Web-приложения, сохраняя в памяти часто используемые данные или данные, требующие значительного времени для получения. Если ваша страница содержит такие данные, то она подходящий кандидат на кэширование.

ASP.NET обеспечивает кэширование с использованием двух основных механизмов: кэширование приложения и кэширование вывода страницы. Первый вариант — это кэширование данных, второй — кэширование готовой страницы.

Кэширование данных и использование переменных приложения похожи тем, что сохраняют информацию в течение всей жизни приложения, однако между ними есть и различия.

Переменные приложения существуют в течение жизни приложения, если только не будут уничтожены явным образом. Для данных, которые сохраняются в кэше, может быть установлено время жизни или оно может быть привязано к каким-то событиям. Переменные приложения больше похожи на глобальные переменные и часто задаются при старте. Кэшированные данные при перезагрузке приложения не восстанавливаются.

Средства управления состоянием на стороне клиента

Такое управление подразумевает, что информация о состоянии должна храниться или на самой странице, или на компьютере клиента. Это также означает, что сервер не хранит никакой информации между посылками данных. Большинство Web-приложений в качестве средства управления на стороне клиента используют куки.

Куки

Это небольшие наборы данных, которые хранятся либо в виде текстового файла на клиентском компьютере, либо в памяти браузера клиента. Сервер вместе со страницей посылает клиенту куки с информацией, относящейся к этому клиенту.

Вы можете использовать куки для хранения информации о клиенте, сеансе или приложении. Компьютер клиента сохранит куки, и, когда браузер запросит страницу, вместе с запросом будет отправлена информация из куки. Сервер может ее прочитать и извлечь любое необходимое значение.

Каждый куки содержит информацию о домене сайта, его создавшего. Вы можете иметь несколько куки, изданных одним и тем же доменом, например его разными поддоменами.

Вы можете создать куки, используя коллекцию Cookies объекта **Response**, например:

```
HttpCookie colorHttpCookie = new HttpCookie("Colors");
```

Далее можно заполнить ее информацией в стиле "ключ-значение":

```
// Добавить значения цветов
colorHttpCookie.Values.Add("Chrome", "Silver");
colorHttpCookie.Values.Add("Foreground", "Black");
colorHttpCookie.Values.Add("Background", "White");
```

После этого мы его помещаем в коллекцию куки объекта **Response**:

```
Response.Cookies.Add(colorHttpCookie);
```

Типы куки

Существуют два типа куки: временные (так называемые сеансовые или непостоянные куки) и постоянные.

- *Временные* куки существуют только в памяти браузера. Когда пользователь выходит из браузера, хранившиеся в нем куки теряются. В предыдущем примере мы создали именно такой куки.
- *Постоянные* куки похожи на временные, однако имеют четко определенный период жизни. Когда запрашивается страница, которая создает постоянный куки, браузер сохраняет его на жестком диске. Вы можете создать постоянный куки, который будет храниться на компьютере клиента месяцы или даже годы. Рассмотрим пример создания куки с временем жизни один час:

```
colorHttpCookie.Expires = DateTime.Now.AddHours(1);
```

Постоянные куки часто используются для хранения информации об имени пользователя, его идентификаторе, чтобы сервер мог его узнать при следующем обращении.

Можно, к примеру, создать куки со временем жизни 100 лет, однако клиент всегда может вручную удалить его со своего компьютера. Так что нет никакой гарантии, что постоянный куки проживет заданный временной период.

Куки приходят на сервер в заголовке HTTP, прочитать их можно через коллекцию Cookies объекта **Request**. Ниже приведен пример получения куки по имени:

```
// Извлечь куки Colors
HttpCookie colorHttpCookie = Request.Cookies["Colors"];
```

Далее извлекаем значения по ключам:

```
// Извлекаем значения
System.Drawing.Color chromeColor =
                System.Drawing.Color.FromName(
                        colorHttpCookie["Chrome"]);
System.Drawing.Color foregroundColor =
                System.Drawing.Color.FromName(
                        colorHttpCookie["Foreground"]);
System.Drawing.Color backgroundColor =
                System.Drawing.Color.FromName(
                        colorHttpCookie["Background"]);
```

View State

Это специальный скрытый контрол, доступный через свойство Page.ViewState. По своей природе он является словарем, предназначенным для сохранения значений между многочисленными запросами одной и той же страницы, что позволяет сохранять ее состояние и настройки.

Перед обращением к серверу текущее состояние страницы и контролов кэшируется в строку, которая помещается в скрытое поле или скрытые поля, их количество зависит от значения свойства Page.MaxPageStateFieldLength. При обратной посылке сервер может по этой информации восстановить состояние страницы. Следующий пример показывает, как можно загружать и получать значения из **ViewState**:

```
// Загрузить значение
ViewState["SelectedCountry"] = "Croatia";
// Прочитать значение
string selectedCountry =
                  (string)ViewState["SelectedCountry"];
```

Перед доступом к данным лучше проверить их наличие, чтобы избежать исключений из-за использования нулевых ссылок:

```
if (ViewState["SelectedCountry"] != null)
```

При желании отключить использование **ViewState** на уровне страницы можно через атрибут EnableViewState директивы Page:

```
<%@ Page... EnableViewState="false" %>
```

Но если вы это сделаете, вы не сможете его включить для отдельных контролов. Чтобы иметь такую возможность, следует задействовать атрибут ViewStateMode, как в следующем примере:

```
<%@ Page... EnableViewState="true"
                                ViewStateMode="false" %>
    ...
<asp:GridView... ViewStateMode="true" />
```

Control State

В ряде случаев может потребоваться для правильной работы контролов сохранять информацию об их состоянии. Например,

на вашем заказном контроле есть несколько вкладок и вам нужно запоминать, какая вкладка открыта. Для этого можно использовать свойство ViewState, однако его можно отключить на уровне страницы.

Поэтому ASP.NET предоставляет вам свойство ControlState, которое позволяет постоянно сохранять информацию, относящуюся к контролу и которое нельзя отключить, как ViewState. Обычно это свойство используется разработчиками применительно к своим заказным контролам.

Скрытые поля

Вы можете также сохранять информацию в контроле **HiddenField**, который при рендеринге превращается в скрытое поле HTML. Оно, конечно, не отображается браузером, зато вы можете устанавливать его свойства, как и у любого другого контрола. Когда страница обращается к серверу, содержимое скрытых полей отправляется туда так же, как и значения прочих контролов. Их можно использовать как хранилище любой информации, относящейся к странице, и которую вы хотите сохранять непосредственно на ней.

Контрол **HiddenField** сохраняет единственное значение в свойстве Value, и оно должно быть явно добавлено в разметке, например, так:

```
<asp:HiddenField id="ExampleHiddenField"
                 value="Example Value" runat="server"/>
```

Раздел 2. Управление пользовательскими данными

Информация о состоянии часто включает в себя данные, уникальные для пользователя, который просматривает ваш Web-сайт. Этот тип информации можно использовать, чтобы подстроить Web-приложение под пользователя (персонализировать). В то же время персонализация включает в себя множество элементов: узнавание пользователя, получение и сохранение информации о нем.

Профили ASP.NET позволяют вам управлять такой информацией, включая и хранение ее в базе данных под управлением SQL-сервера.

Что такое профили ASP.NET?

Некоторые Web-сайты устроены так, что им нужно определить, прошел ли пользователь аутентификацию или он является анонимным. Такие сайты сохраняют как минимум кусочек информации, имеющей отношение к пользователю, хотя зачастую этой информации больше: это может быть выбранная цветовая схема сайта, имя пользователя, адрес, что-то еще. Учет его персональных настроек побуждает пользователя чаще заходить на такой Web-сайт. В то же время добавление персонализации с использованием той или иной технологии управления состоянием требует изрядной работы, так что использование встроенных возможностей работы с профилями ASP.NET может сберечь время разработчика. Перечислим основные преимущества такого подхода.

- Профили ASP.NET позволяют сохранять информацию о пользователе в виде свойств. Данные профиля сохраняются в базе данных под управлением SQL-сервера и ассоциируются с конкретным пользователем.
- Профили ASP.NET обеспечивают нам обобщенную систему хранения, которая позволяет определять и поддерживать почти любой тип данных с использованием заказных свойств.
- Профили ASP.NET позволяют вам даже не заботиться о создании и поддержке вашей базы данных, поскольку этим всем занимается специальная утилита ASP.NET SQL Server Registration Tool.

Чтобы использовать этот механизм, вам нужно будет указать провайдера профиля в файле конфигурации Web.config. Также потребуется создать реальные (действующие в данном случае) свойства профиля. Для этого просто надо объявить эти свойства в Web.config с использованием элемента profile и его дочерних элементов.

Вам не нужно явно определять, кто является текущим пользователем, или выполнять циклические действия для поиска значения в базе данных, достаточно просто выполнить установку значения свойства или его получения, а обо всем остальном позаботится ASP.NET.

Эти возможности становятся доступными за счет использования строго типизированного прикладного интерфейса пользователя (API), который доступен отовсюду из вашего приложения через свойство Profile класса **HttpContext**.

Использование профилей ASP.NET

Чтобы начать использование профилей ASP.NET, их нужно сначала установить, а затем активировать.

Установка базы данных

Базу данных нужно создать, для этого можно воспользоваться утилитой ASP.NET SQL Server Registration Tool или попросить Visual Studio сделать это для вас. Если вы хотите использовать базу данных SQL Server для хранения профиля, в командной строке Visual Studio нужно выполнить команду

```
aspnet_regsql -S MY_DATABASE -E -A p
```

Здесь опция "-E" означает, что нужно использовать Windows-полномочия пользователя, указанного при входе в систему, опция "-S" задает имя базы данных MY_DATABASE, значение "–A p" добавляет хранилище профиля. База данных не привязана к конкретному Web-приложению, она может быть использована произвольным количеством приложений и пользователей.

Установка приложения

Для записи и чтения данных профиля из соответствующей базы данных вы можете использовать профиль по умолчанию, прописанный в файле machine.config. Это **AspNetSqlProvider** — экземпляр класса **SqlProfileProvider**. Он работает в связке с SQL-сервером, установленным на локальном компьютере. Вы можете использовать его, но можете и заменить его другим, сделав соответствующую запись в Web.config, например, такую:

```
<system.web>
...
    <profile defaultProvider="SqlProfileProvider">
      <providers>
        <add name="SqlProfileProvider"
          type="System.Web.Profile.SqlProfileProvider"
          connectionStringName=
            "customerManagementASPNETConnectionString"
```

```
            applicationName="CustomerManagement"
            description="My Profile Provider " />
        </providers>
    </profile>
...
</system.web>
```

В этом примере атрибут defaultProvider элемента profile указывает, какого поставщика использовать, если не указан другой. В элементе add атрибут name определяет конкретного поставщика профиля, это имя можно использовать из кода. Атрибут type указывает имя класса, обеспечивающего функциональность поставщика.

Атрибут connectionStringName ссылается на строку коннекции, определенную в элементе connectionStrings из файла Web.config.

Атрибут applicationName указывает имя вашего Web-приложения, что дает возможность использовать более одного приложения с одной и той же базой данных для профилей.

Атрибут description содержит описание поставщика, что облегчает его идентификацию, если имеется несколько определений провайдера.

Вы также можете определить свойства действующего профиля через использование профиля ASP.NET. Вы просто даете в файле Web.config определение свойств в элементе profile и его дочерних элементах, например:

```
<profile>
    <providers>
    ...
    </providers>
    <properties>
        <add name="FirstName" type="string" />
        <add name="LastName" type="string" />
    </properties>
</profile>
```

Элемент properties содержит список свойств, в нашем примере это свойства FirstName и LastName.

При запуске приложения ASP.NET создает динамически определяемый класс **ProfileCommon**, который является наследником класса **System.Web.Profile.ProfileBase**. Класс **ProfileCommon**

включает в себя свойства, создаваемые из определений свойств профиля, указанного в конфигурации приложения.

В следующем примере кода показаны свойства, которые автоматически создаются из определений свойств профиля.

```
public class ProfileCommon :
                          System.Web.Profile.ProfileBase
{
    public virtual string LastName
    {
        get
        {
           return ((string)
                  (this.GetPropertyValue("LastName")));
        }
        set
        {
            this.SetPropertyValue("LastName", value);
        }
    }
    public virtual string FirstName
    {
        get
        {
            return ((string)
                (this.GetPropertyValue("FirstName")));
        }
        set
        {
            this.SetPropertyValue("FirstName", value);
        }
    }
    public virtual ProfileCommon GetProfile(
                                        string username)
    {
        return ((ProfileCommon)
                        (ProfileBase.Create(username)));
    }
}
```

Далее экземпляр динамического класса **ProfileCommon** устанавливается как значение свойства Profile текущего экземпляра класса **HttpContext**. Это делает пользовательскую информацию

доступной из любого места приложения посредством строго типизированного API. Следующий пример показывает, как можно получить доступ к этому свойству:

```
ProfileBase userProfile = HttpContext.Current.Profile;
```

Чтобы получить определение класса **ProfileBase,** вы обязаны импортировать пространство имен **System.Web.Profile**. На Web-форме вы также можете получить доступ к профилю текущего пользователя через свойство Profile класса **Page**.

Приведем пример того, как в коде можно использовать значения свойств определенного вами профиля:

```
// Задать значение свойства FirstName
Profile.FirstName = "Gregory";
// Получить значение свойства FirstName
string firstName = Profile.FirstName;
```

При записи значения в свойство профиля оно автоматически сохраняется для текущего пользователя, от вас не требуется писать дополнительный код для обращения к базе данных.

Раздел 3. Управление кэшированием

Для создания высокопроизводительных масштабируемых Web-приложений очень важна возможность хранения в памяти объектов, страниц или частей страниц, то есть реализация кэширования. В качестве места хранения может выступать Web-сервер, прокси-сервер или браузер. Организация кэширования позволяет не получать повторно ранее созданную полученную информацию.

Механизмы кэширования ASP.NET позволяют легко сохранять между HTTP-запросами результаты вычислений или целые страницы.

Механизмы кэширования ASP.NET

Большое количество Web-приложений работает с данными, которые требуют значительного времени для своего получения или вычисления, а значит, есть необходимость хранения часто используемых данных.

ASP.NET предоставляет нам два основных механизма для этой цели:

- *кэширование приложения*, например значений, полученных из базы данных;
- *кэширование вывода страницы*, чтобы не создавать заново разметку, отправляемую браузеру.

Кэширование приложения использует хранение данных в памяти в виде пары "ключ-значение". Это похоже на хранение информации о состоянии приложения, однако, в отличие от последнего, данные в кэше не хранятся в течение всего жизненного цикла приложения. ASP.NET управляет кэшем и удаляет данные, когда истечет срок их хранения, когда они становятся недействительными или когда не хватает памяти. Этот механизм вы можете настроить так, чтобы получать уведомления при удалении элемента.

Кэш вывода страницы сохраняет в памяти содержимое обработанной страницы, что позволяет не запускать заново процесс ее создания. Это особенно полезно для страниц, которые не слишком часто меняются и в то же время требуют значительных затрат на свое создание, к примеру, для страниц, отображающих иногда изменяющиеся данные. Кэширование вывода можно настроить отдельно для каждой страницы или создать профиль кэширования в файле Web.config. Создание профилей кэша позволяет вам, задав однажды параметры кэширования, использовать их на нескольких страницах.

Использование кэширования ASP.NET

Кэш приложения предлагает способ хранения информации в виде пары "ключ-значение".

Как было сказано ранее, механизм ASP.NET берет на себя управление кэшем и удаляет данные оттуда при выполнении любого из следующих условий:

- истекло время хранения;
- данные устарели;
- не хватает памяти.

Замечание. Механизм всегда при запросе данных проверяет их наличие. Если они есть, то выдаются сразу, если нет, то создаются заново и помещаются в кэш.

Механизм кэширования обеспечивается классом **Cache** из пространства имен **System.Web.Caching**. Экземпляр этого

класса является собственностью приложения и разделяет его жизненный цикл, то есть при перезапуске кэш создается заново.

Данные кэша приложения

Рассмотрим типичные действия при хранении данных в кэше приложения на примере объекта **DataSet**:

```
// Создать DataSet
System.Data.DataSet customerDataSet =
                                new System.Data.DataSet();
// Заполнить DataSet
populateDataSet(ref customerDataSet);
// Поместить в кэш
Cache["CustomerDataSet"] = customerDataSet;

// Получить из кэша
customerDataSet = (System.Data.DataSet)
                                 Cache["CustomerDataSet"];
if (customerDataSet == null)
{
    // Заполнить DataSet
    populateDataSet(ref customerDataSet);
    // Поместить в кэш
    Cache["CustomerDataSet"] = customerDataSet;
}
```

Обратите внимание на проверку наличия элемента в кэше при его получении. Так мы избегаем выбрасывания исключений при использовании нулевых ссылок. Далее будет показано, как для добавления объектов в кэш можно использовать методы Add и Insert класса **Cache**.

Класс **Cache** позволяет настроить правила удаления элементов из кэша, это еще называется очисткой (scavenging). Вы можете установить приоритет одних данных перед другими при выполнении очистки. Для этой цели можно при добавлении данных посредством методов Add или Insert указать одно из перечисляемых значений типа **CacheItemPriority**:

```
// Поместить объект в кэш, если он там был, то перезаписать
Cache.Insert("CustomerDataSet", customerDataSet, null,
             Cache.NoAbsoluteExpiration,
             Cache.NoSlidingExpiration,
             CacheItemPriority.Default, null);
```

```
// Поместить объект в кэш, если его там не было
Cache.Add("CustomerDataSet", customerDataSet, null,
            Cache.NoAbsoluteExpiration,
            Cache.NoSlidingExpiration,
            CacheItemPriority.Default, null);
```

В этих примерах использован приоритет Default. Для большей выживаемости элемента можно было бы указать значение High.

Разница между методами Add и Insert заключается в том, что при наличии объекта в кэше метод Insert его перезаписывает. Метод Add добавляет объект, только если его в кэше не было.

При задании политики удаления можно установить время жизни с момента последнего обращения либо абсолютное время, когда следует удалить элемент. В следующем примере кода указывается, что объект подлежит удалению 1 января 2022 года:

```
// Объект подлежит удалению 1 января 2022 года
Cache.Insert("CustomerDataSet", customerDataSet, null,
            new DateTime(2022, 1, 1),
            Cache.NoSlidingExpiration,
            CacheItemPriority.Default, null);
```

А здесь мы устанавливаем время жизни в один час с момента последнего обращения:

```
// Время жизни — 1 час с момента последнего обращения
Cache.Insert("CustomerDataSet", customerDataSet, null,
            Cache.NoAbsoluteExpiration,
            new TimeSpan(1, 0, 0),
            CacheItemPriority.Default, null);
```

Также можно установить зависимость времени жизни элемента (или его правильности) от внешнего файла или директории или даже от другого элемента кэша. Если они изменяются, то элемент из кэша удаляется.

Далее показано, как можно установить зависимость элемента от объекта **CountryDataSet** из кэша (первый пример) или от изменения файла Countries.xml (второй пример):

```
// Помещаем в кэш объект countryDataSet
Cache.Insert("CountryDataSet", countryDataSet, null,
            Cache.NoAbsoluteExpiration,
            Cache.NoSlidingExpiration,
            CacheItemPriority.Default, null);
```

```
// Массив ключей зависимости
string[] cacheKeyDependency = { "CountryDataSet" };

// Помещаем в кэш объект customerDataSet
// с указанием зависимости от countryDataSet
Cache.Insert("CustomerDataSet", customerDataSet,
             new CacheDependency(null,
                                 cacheKeyDependency),
             Cache.NoAbsoluteExpiration,
             Cache.NoSlidingExpiration,
             CacheItemPriority.Default,
             new CacheItemRemovedCallback(
                             dataSetRemovedCallback));

// Помещаем в кэш объект customerDataSet
// с указанием зависимости от файла Countries.xml
// в корневой папке приложения
Cache.Insert("CustomerDataSet", customerDataSet,
             new CacheDependency(Server.MapPath(
                                 "~/Countries.xml")),
             Cache.NoAbsoluteExpiration,
             Cache.NoSlidingExpiration,
             CacheItemPriority.Default,
             new CacheItemRemovedCallback(
                             dataSetRemovedCallback));
```

В этих примерах последний параметр задает метод обратного вызова для обработки уведомления об изменении кэша, его стандартное оформление имеет примерно такой вид:

```
public void dataSetRemovedCallback(String key,
    object value, CacheItemRemovedReason removedReason)
{
    // Наша реакция на удаление объекта из кэша
    ...
}
```

Кэширование вывода страниц

Этот механизм ASP.NET позволяет кэшировать все или некоторые из ранее сгенерированных ответов на запросы клиента. Кэшировать можно в браузере, сделавшем запрос, на Web-сервере, его выполняющем, или на другом устройстве, позволяющем это сделать, например прокси-сервере.

Кэширование вывода может существенно улучшить производительность Web-приложений. Вы можете задать параметры кэширования или декларативно на странице, или в файле конфигурации, или программным путем посредством специального API кэширования.

Для декларативной настройки нужно на страницу поместить директиву OutputCache, где задать значения атрибутов Duration и VaryByParam:

```
<%@ OutputCache Duration="30" VaryByParam="None"%>
```

В нашем примере страница помещается в кэш на 30 секунд, причем существует только одна версия страницы, на что указывает значение None атрибута VaryByParam.

Атрибут Duration является обязательным. Кроме него, в директиве OutputCache вы обязательно должны использовать хотя бы один из следующих атрибутов.

- VaryByContentEncoding. Это разделенный запятыми список, задающий кодировку содержимого кэша.
- VaryByControl. Набор идентификаторов контролов (через точку с запятой), которые содержатся на странице или пользовательском элементе управления и используются для изменения текущей записи кэша.
- VaryByCustom. Список пользовательских строк, которые используются для изменения записи кэша.
- VaryByHeader. Набор имен заголовков (через запятую) для изменения записи кэша. Они задают заголовки HTTP соответствующего запроса.
- VaryByParam. Список строк запроса или параметров POST (через запятую), от которых зависит состояние кэша.

Вы можете использовать кэширование страниц, основанное на значениях параметров из строки запроса, значениях переменных формы или контролов. Кэширование, основанное на значениях, должно быть прописано в атрибутах VaryByParam или VaryByControl директивы OutputCache.

Когда пользователь запрашивает закэшированную страницу, ASP.NET определяет, является ли она валидной (правильной, точнее, действующей) на основе политики кэширования, которую вы задали для страницы. Если это так, то страница из кэша

отправляется клиенту без дополнительной обработки. ASP.NET позволяет запускать код в процессе этой проверки, так что вы можете реализовать собственную логику проверки валидности страницы.

В ряде случаев нецелесообразно кэшировать страницу целиком. Скажем, если она содержит часто изменяемые фрагменты, то есть смысл кэшировать только те части страниц, которые изменяются редко. ASP.NET обеспечивает функциональность и для этого варианта.

Лекция 14. Безопасность ASP.NET Web-приложений

Безопасность Web-приложения — один из ключевых вопросов его разработки. Разработка безопасной системы требует тщательного планирования. При этом и администраторы, и разработчики должны иметь ясное понимание имеющихся в их распоряжении средств и возможностей. Web-приложение настолько безопасно, насколько безопасно его самое слабое звено.

Раздел 1. Обзор вопросов безопасности

Забота о безопасности Web-приложения объясняется еще и тем, что оно имеет доступ к центральному Web-серверу системы и прочим серверам, включая серверы баз данных.

В данном разделе содержится обзор основных концепций безопасности, таких как аутентификация и авторизация, методов аутентификации и механизмов ее проведения.

Что такое аутентификация и авторизация?

Аутентификация — это процесс получения учетных данных, таких как имя пользователя и пароль, а также проверки, что учетный данные верны, например, путем сравнения с хранящимися в базе данных значениями.

Аутентификация может проводиться не только в отношении пользователей, но и в отношении служб, процессов, компьютеров. Все они обязаны для работы в сети предоставить имя и пароль. Затем эти значения проверяются по базе данных или с использованием службы Active Directory Domain Services (AD DS).

Авторизация. После того как пользователь прошел проверку подлинности, процесс авторизации определяет, имеет ли тот право доступа к запрашиваемому ресурсу. Авторизация может либо отказать в доступе аутентифицированному пользователю, либо разрешить доступ.

Например, вы можете доступ к защищенным страницам одним пользователям разрешить, а другим запретить.

Надлежащим образом организованные аутентификация и авторизация позволяют уменьшить риски при использовании

вашего приложения, причем планировать их следует уже на ранних стадиях разработки.

Доменные службы Active Directory предоставляют вам средства для управления этими процессами в сети вашей организации, впрочем, эти вопросы относятся уже к компетенции администраторов.

Методы аутентификации ASP.NET

ASP.NET обеспечивает два варианта проведения аутентификации: на основе аутентификации при входе в Windows и аутентификацию, основанную на использовании форм. Оба варианта предполагают наличие кода, необходимого для проверки подлинности учетных данных пользователя.

Windows-аутентификация

Она основана на проверке подлинности, которая осуществляется при входе пользователя в Windows. ASP.NET использует ее вместе с IIS-аутентификацией.

Когда пользователь запрашивает защищенную страницу Web-приложения, запрос проходит через IIS. Если учетные данные пользователя не позволяют его авторизовать для получения запрошенного ресурса, IIS отклоняет запрос. Далее пользователю предоставляется форма для проведения авторизации (ввести имя и пароль). После ее заполнения IIS опять проверяет данные, и если те верны, то направляет исходный запрос Web-приложению, а уже оно формирует и отправляет клиенту защищенную страницу.

Аутентификация, основанная на формах

В этом варианте неавторизованные запросы направляются на HTML-форму с использованием механизма HTTP-перенаправления на стороне клиента. Пользователь заполняет и отправляет форму. Система после проверки информации выдает специальные куки аутентификации.

Последующие запросы пользователей уже будут в своем заголовке содержать эти куки, и этого будет достаточно для прохождения аутентификации.

Поставщик форм аутентификации предоставляет информацию о том, как создать форму для конкретного приложения и как провести процесс аутентификации с использованием своего кода.

Удобным вариантом работы с данным типом аутентификации является использование специальных контролов ASP.NET, которые обеспечивают способ получения учетных данных пользователей, проверку их подлинности и управления ими, требуя минимального объема ручного кодирования или позволяя обойтись совсем без него.

Сценарии поддержки методов аутентификации ASP.NET

Оба помянутых варианта имеют свои достоинства и свои недостатки, так что выбор нужно делать в зависимости от сценария работы вашего Web-приложения.

Применение Windows-аутентификации

Этот вариант обычно используется в интрасети. Он использует существующую инфраструктуру Windows, поэтому лучше всего подходит для ситуаций, когда у вас есть определенное количество пользователей с существующими учетными записями.

В тех случаях, когда требуется программным путем добавлять учетные записи пользователей, этот вариант практически непригоден.

Применение аутентификации, основанной на формах

Этот вариант является более подходящим решением в случае, если вы хотите создать собственную систему регистрации пользователей на Web-узле. Он подходит для большинства Интернет-приложений.

Преимуществом этого типа аутентификации является то, что он позволяет хранить имена пользователей и пароли с использованием того механизма и способа хранения, которые вы выбираете сами. Например, вы можете хранить их в файле конфигурации, XML-файле или таблице базы данных.

Этот вариант для идентификации пользователя обычно использует куки. Пока пользователь этот куки не получит, он не сможет получить доступ к запрашиваемой защищенной странице,

так как ASP.NET будет перенаправлять его на страницу для ввода учетных данных.

Механизмы аутентификации IIS

Как уже отмечалось ранее, Web-приложения ASP.NET обычно запускаются под управлением службы IIS (Internet Information Services). Подробное рассмотрение вопросов ее настройки выходит за рамки нашего учебного курса, тем более что в настоящее время на разных компьютерах с разными версиями Windows могут быть установлены разные версии IIS. Кроме того, для осуществления таких настроек требуются права администратора, которые в учебном классе студентам, очевидно, не предоставляются. Впрочем, необходимые детальные инструкции по настройке установленной у вас версии IIS несложно найти в сети Интернет. Поэтому здесь мы рассмотрим только общие моменты механизмов проведения аутентификации.

Чтобы использовать аутентификацию, основанную на проверке подлинности Windows, в IIS следует отключить анонимный доступ. В этом случае, если пользователь запрашивает защищенную страницу, аутентификацию будет производить IIS.

Вообще-то, IIS предоставляет несколько механизмов для этой цели, такие как:

- анонимный доступ;
- базовая аутентификация;
- аутентификация, основанная на сертификате клиента;
- сборная аутентификация;
- аутентификация, основанная на IIS-сертификате;
- аутентификация, встроенная в Windows.

Не обязательно в вашей версии IIS вы сможете использовать все эти варианты. Например, IIS 7 версии по умолчанию поддерживает только первый и последний из них.

Анонимный доступ

Рассмотрим типичную ситуацию. Открытое Web-приложение позволяет неизвестным пользователям выполнять запросы. На такой случай в IIS предусмотрена поддержка анонимных пользователей, то есть тех, которые не имеют учетных данных или не предоставляют их.

Когда IIS получает запрос от анонимного пользователя, он передается операционной системе Windows с использованием (по умолчанию) эккаунта IUSR. Этот вариант является для IIS вариантом аутентификации по умолчанию.

Базовый метод проверки подлинности

В этом варианте пользователю, не имеющему необходимых полномочий, будет предложено ввести свои учетные данные. Эта информация передается службе IIS, а там уже она становится доступной Web-приложению.

К преимуществам базовой аутентификации можно отнести и тот факт, что она является частью спецификации HTTP и поддерживается большинством браузеров.

Этот вариант является самым обычным способом ограничения доступа к Web-приложению, однако он не очень безопасен, поскольку имя пользователя и пароль передаются службе IIS как обычный текст. В целях повышения безопасности можно использовать протокол защищенных сокетов SSL для шифрования учетных данных при передаче по сети.

Аутентификация на основе проверки сертификатов клиентов

Этот вариант предполагает, что пользователь при входе предъявляет клиентский сертификат, что делает излишней аутентификацию посредством каких-либо других методов. Если вы хотите использовать доменные службы Active Directory для проведения аутентификации таких клиентов, вам нужно соответствующим образом настроить AD DS. Отметим здесь, что в этом случае вы не можете использовать сертификацию IIS.

Сборная проверка

Сборная проверка (digest authentication) похожа на базовую, однако использует шифрование при отправке пользовательской информации на сервер.

Если анонимный доступ отключен, то пользователям предлагается ввести учетные данные. Браузер объединяет эту информацию с другой информацией на стороне клиента и отправляет ее на сервер в виде закодированного хэша, так

называемого Message Digest5 (MD5). Сервер для восстановления этой информации использует собственный хэш.

Замечание. Для задействования этого типа аутентификации требуется, чтобы браузер поддерживал протокол HTTP 1.1, впрочем, это можно сказать обо всех современных браузерах. Анонимную аутентификацию в этом случае требуется отключить. При первом запросе все браузеры пытаются выполнить доступ к ресурсу анонимно; если этого не запретить, то пользователь может получить доступ к секретной информации.

Аутентификация, основанная на клиентских сертификатах IIS

Здесь могут использоваться два метода сопоставления клиентских сертификатов IIS: "один-к-одному" и "один-ко-многим".

Метод "один-к-одному" предполагает, что каждому пользователю Windows соответствует один клиентский сертификат. Сертификат, посылаемый клиенту, должен быть идентичен копии сертификата, хранящейся на сервере. В целях безопасности на своем Web-сайте необходимо включить SSL.

Метод "один-ко-многим" используется, если следует убедиться, что сертификат клиента содержит конкретную информацию, скажем об эмитенте или субъекте. При использовании этого варианта принимаются все клиентские сертификаты, которые удовлетворяют определенным критериям. В этом случае также необходимо на своем узле включить SSL.

Встроенная в Windows аутентификация

Если пользователь, который уже прошел процесс аутентификации в Windows, делает запрос, IIS может использовать его учетные данные при доступе к ресурсу. При этом мандат пользователя не содержит ни имени, ни пароля, а только лишь зашифрованный токен, который подтверждает, что пользователь обладает безопасным статусом.

Этот вариант аутентификации есть смысл использовать, если вы хотите, чтобы клиенты прошли проверку подлинности с помощью протоколов NTLM или Kerberos. Впрочем, более подробное рассмотрение данного вопроса выходит за рамки нашего учебного курса.

Аутентификация и авторизация, основанные на формах

Это самый распространенный вариант. Вы можете реализовать основанную на формах аутентификацию и авторизацию как для неаутентифицированного, так и для аутентифицированного клиента.

Процесс выглядит следующим образом.

1. Когда неаутентифицированный клиент запрашивает защищенную страницу, запрос проходит через IIS.
2. IIS должен быть настроен на разрешение анонимного доступа с использованием аутентификации, основанной на формах, поэтому запрос направляется на модуль ASP.NET, отвечающий за такую аутентификацию.
3. ASP.NET проверяет, имеет ли клиент аутентификационный куки. Поскольку это первый визит, то куки нет, поэтому клиента перенаправляют на страницу для ввода учетных данных.
4. Там он вводит свое имя и пароль.
5. Эта информация проверяется, как правило, по базе данных пользователей.
6. Если полномочия не признаны, то клиенту отказывают в доступе.
7. Если все в порядке, создается аутентификационный куки, а клиент получает право доступа к запрошенной странице и перенаправляется на нее.
8. Когда аутентифицированный клиент запрашивает страницу, к его запросу прицепляется ранее созданный аутентификационный куки.
9. Запрос сначала проходит через IIS, а затем на модуль ASP.NET, отвечающий за аутентификацию на формах.
10. Этот модуль проверяет куки и, если все в порядке, разрешает доступ к странице.

Протокол защищенных сокетов

IIS предоставляет пользователям защищенный канал связи за счет поддержки протокола SSL и шифрования данных на сервере и на стороне клиента.

SSL — это коммуникационный протокол для безопасной передачи данных в сети. Для этой цели он использует:

- шифрование данных;
- аутентификацию сервера;
- проверку целостности данных.

Шифрование данных

При передаче на сервер информации, введенной на форме, она может быть перехвачена. Поэтому SSL шифрует информацию при передаче данных из браузера на сервер и обратно.

Информация шифруется с использованием открытого алгоритма и сеансового ключа. Web-сервер генерирует открытый ключ, который может использоваться любым клиентом. Клиент генерирует ключ сеанса и шифрует его посредством открытого ключа перед передачей на сервер. Далее данные шифруются с использованием этого сеансового ключа.

Сила шифрования определяется длиной ключа. IIS поддерживает шифрование с длиной ключа от 40 до 128 бит. Ее можно задать в файле конфигурации Web.config:

```
<system.webServer>
    ...
    <security>
        <access sslFlags="Ssl128"/>
    </security>
</system.webServer>
```

Аутентификация сервера

Это процесс проверки подлинности сервера. Он гарантирует, что данные передаются на правильный сервер и что сервер безопасен.

Если вы что-то покупаете в сети, вы вводите данные своей банковской карточки. Злоумышленники могут подсунуть вам фальшивый Web-сайт, чтобы украсть ваши данные.

В процессе аутентификации сервера тот посылает клиенту сертификат, подтверждающий его подлинность. Для предотвращения подмены сайта вы можете установить протокол SSL на своем Web-сервере. А это, в свою очередь, требует установки сертификата сервера. Сертификат содержит информацию о вашей организации, вашем Web-сайте и эмитенте сертификата.

Сертификат должен быть подписан центром сертификации, который выступает в качестве доверенной третьей стороны, проверяющей подлинность Web-сайта для своих пользователей.

SSL также поддерживает клиентские сертификаты для аутентификации. Они используются для проверки уже не Web-серверов, а Web-браузеров.

Проверка целостности данных

SSL защищает целостность данных при передаче между сервером и браузером. SSL гарантирует, что при передаче данные ни в коем случае не будут изменены. При передаче в сообщения включается специальный код проверки подлинности, он позволяет определить, было ли сообщение изменено.

После настройки сервера и Web-сайта на применение SSL доступ к страницам можно получать с использованием безопасного соединения. Для получения страницы SSL использует безопасный HTTP (HTTPS). В запросе будет наличествовать префикс https:// вместо http://. Этот формат будет использоваться для получения любой Web-страницы с вашего Web-узла. Портом по умолчанию в этом случае является 443.

Замечание. ASP.NET позволяет программно определить, осуществляется ли связь по протоколу HTTPS через свойство Request.IsSecureConnection.

Раздел 2. Декларативная настройка аутентификации и авторизации

Для защиты своих Web-приложений вы можете не только использовать аутентификацию и авторизацию, но и распознавать и отслеживать, какие пользователи получают доступ к вашему сайту.

Процесс аутентификации, основанный на использовании форм или проверке Windows, вы можете настроить через конфигурирование IIS или путем добавления в файл Web.config соответствующих правил аутентификации и авторизации.

Далее в этом разделе мы рассматриваем именно такой декларативный способ обеспечения безопасности Web-приложений или конкретных ресурсов.

Настройка аутентификации, встроенной в Windows

Обеспечение безопасности через проверку подлинности Windows состоит из двух шагов:

1. Включение этого варианта в настройках IIS;
2. Внесение соответствующих установок в файл Web.config.

Настройка IIS

Нужно в настройках отключить анонимную аутентификацию и включить один или несколько других (из вариантов Basic и Integrated Windows можно выбрать только один).

В случае выбора варианта Integrated Windows нельзя использовать брандмауэр или прокси-сервер.

Установки в Web.config

Если вы используете Windows-аутентификацию, нужно установить этот вариант в Web.config. Для этого предназначены секции authentication, authorization и identity. Приведем пример того, как в Web.config назначить использование аутентификации Windows:

```
<system.web>
    <authentication mode="Windows" />
</system.web>
```

Вы также можете использовать элемент identity, чтобы включить имперсонализацию, что позволит серверу исполнять код в контексте безопасности конкретной личности или анонимного пользователя. Имперсонализация в ASP.NET не является обязательной и по умолчанию отключена. Использовать элемент identity можно либо в файле Web.config, либо в файле Machine.config, примерно так:

```
<identity impersonate="true|false"
    username="username"
    password="password" />
```

В этом примере атрибуты username и password определяют учетную запись пользователя, если имперсонализация включена. Здесь они указываются в незашифрованном виде.

Если имперсонализация выключена, а по умолчанию это так, то используется специальная учетная запись. Для Windows 7 это Имя_компьютера\w3wp.

Настройка аутентификации, основанной на формах

Этот вариант подразумевает выполнение трех шагов:

1. Включить аутентификацию, основанную на формах, в настройках IIS;
2. В файл Web.config добавить соответствующий элемент authentication;
3. Создать форму, предназначенную для ввода учетных данных пользователя.

Для выполнения первого шага включаем анонимную аутентификацию, так что проверка подлинности будет осуществляться не IIS, а ASP.NET.

Настройки процесса аутентификации в Web.config прописываются в секциях authentication и authorization:

```
<system.web>
    <authentication mode="Forms">
       <forms name=".ASPXAUTH" loginUrl="Login.aspx" />
    </authentication>
</system.web>
```

Здесь вы также можете настроить параметры использования куки, предназначенных для хранения информации о пользователе. Можно задать суффикс, используемый в именах куки и URL страницы, на которую следует перенаправлять неаутентифицированные запросы. В приведенном примере мы употребили значения, используемые по умолчанию.

Ну и наконец, вы должны создать Web-форму для ввода пользователем своих учетных данных.

Настройка авторизации

После включения и настройки процесса аутентификации вы можете настроить параметры безопасности так, чтобы разрешить или запретить доступ к определенному ресурсу, скажем странице или папке, то есть настроить процесс авторизации. Сделать это можно декларативно, указав ресурсы, пользователей и роли в файле Web.config.

Чтобы указать, что определенные страницы или папки являются защищенными, вы можете создать элемент location, содержащий внутри себя элементы system.web и authorization для каждой защищенной страницы или папки. Приведем пример:

```
<location path="ImportCountries.aspx">
    <system.web>
        <authorization>
            <deny users="?" />
        </authorization>
    </system.web>
</location>
```

Все параметры, указанные в элементе location, относятся только к странице или папке, которая указана в атрибуте path.

Разумеется, в конфигурационной секции вы можете иметь несколько элементов location. Кроме того, файл Web.config может располагаться в подпапке и определять параметры авторизации для ее содержимого. Если элемент location относится к папке, он, очевидно, относится ко всем страницам, в ней расположенным.

В элементе system.web располагается элемент authorization, чтобы указать тип авторизации, который вы хотите реализовать. Вы можете использовать элементы allow или deny, чтобы разрешить или запретить пользователям доступ к странице.

Это самозакрывающиеся элементы, где атрибут users определяет пользователей, а атрибут roles — роли. Вопросительный знак (?) означает любого анонимного пользователя, а звездочка (*) — всех пользователей. Рассмотрим пример запрета доступа для всех анонимных пользователей:

```
<authorization>
    <deny users="?" />
</authorization>
```

А это пример разрешения доступа для пользователя Gandalf:

```
<authorization>
    <allow users="Gandalf" />
</authorization>
```

Замечание. Конечно, не слишком разумно авторизовать пользователей индивидуально, так как это конфиденциальная информация, а файл Web.config могут и украсть. Кроме того, такой подход никак нельзя назвать гибким, поскольку он не позволяет изменять эту информацию программным путем во время выполнения. Такой вариант годится разве что для тестирования.

Приведем еще пару примеров. Это запрет доступа к странице всех анонимных пользователей:

```
<location path="ImportCountries.aspx">
    <system.web>
        <authorization>
            <deny users="?" />
        </authorization>
    </system.web>
</location>
```

А это такой же запрет для всего приложения:

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

Раздел 3. Программная настройка аутентификации и авторизации

При работе с декларативно установленной аутентификацией и авторизацией в Web-приложении вы все же имеете возможность программного доступа к информации о пользователе, а также можете использовать для такого доступа Login-контролы.

В данном разделе мы разберем вопросы программной идентификации пользователей, а также научимся создавать Web-формы для проведения аутентификации.

Обзор Login-контролов

ASP.NET предоставляет разработчикам набор специальных элементов управления, которые позволяют легко автоматизировать процесс аутентификации пользователей. Перечислим некоторые контролы из этой категории, которые вы можете найти на панели инструментов Toolbox.

- *ChangePassword* — позволяет пользователю изменить свой пароль;
- *CreateUserWizard* — собирает информацию о новом пользователе и добавляет его в систему членства ASP.NET;

- *Login* — предоставляет все необходимые элементы для аутентификации пользователя;
- *LoginName* — вывод имени пользователя;
- *LoginStatus* — ссылка, ведущая на разные страницы для аутентифицированных и неуатентифицированных пользователей;
- *LoginView* — позволяет отображать разную информацию для аутентифицированных и неуатентифицированных пользователей;
- *PasswordRecovery* — позволяет пользователям восстановить пароль, который будет направлен на почтовый адрес, указанный при создании эккаунта. Требуется сервер с поддержкой протокола SMTP.

Контрол **Login**, будучи помещенным на Web-форму, принимает имя пользователя и пароль, а затем с использованием членства ASP.NET и механизма Form-аутентификации проверяет учетные данные и создает аутентификационный талон.

Замечание. По умолчанию этот процесс настроен на использование страницы с именем Login.aspx. Вы можете назначить вместо нее другую страницу путем установки атрибута loginUrl для контрола **Login**.

Контрол **LoginName** отображает в заданной позиции на всех страницах информацию о пользователе, вошедшем в систему. Этот элемент обычно используется в сочетании с контролом **LoginStatus**.

LoginStatus отображает в определенном месте на всех страницах ссылку для входа или выхода из системы в зависимости от текущего статуса аутентификации. Обычно используется в сочетании с **LoginName**.

После входа пользователя в систему вы можете их перенаправить путем установки свойства DestinationPageUrl контрола **Login**. Если это свойство не установлено, пользователь направляется на изначально запрошенный URL.

Доступ к данным, идентифицирующим пользователя

После проведения аутентификации становится возможным доступ к данным, идентифицирующим пользователя.

Для этого можно использовать объекты, реализующие интерфейс **System.Security.Principal.IPrincipal**. Например, можно получить информацию о конкретном HTTP-запросе и о сделавшем его пользователе. Основной объект этого механизма представляет собой контекст безопасности учетной записи пользователя, от имени которого выполняется код программы.

Класс **System.Web.HttpContext** включает в себя всю информацию о запросе, специфичную для протокола HTTP. Он имеет свойство User, возвращающее объект, реализующий интерфейс **IPrincipal**.

Ниже приведен пример того, как можно сохранить ссылку, полученную через свойство User.

```
System.Security.Principal.IPrincipal requestPrincipal =
                                    HttpContext.Current.User;
```

Отметим здесь, что совместно используемое статическое свойство Current класса **HttpContext** содержит ссылку на текущий контекст HTTP. Это необходимо для доступа к свойству User, который требует наличия экземпляра класса **HttpContext**.

При написании кода Web-формы вы также имеете доступ к этой информации через свойство Page.User:

```
System.Security.Principal.IPrincipal requestPrincipal =
                                              Page.User;
```

Реальный тип этих объектов может быть разным. Объекты, реализующие интерфейс **IPrincipal**, полученные через свойство User, могут иметь тип **System.Security.Principal.WindowsPrincipal** при использовании Windows-аутентификации; в случае аутентификации, основанной на формах, они будут иметь тип **System.Web.Security.RolePrincipal**.

Интерфейс **IPrincipal** определяет только одно свойство, а именно свойство Identity, возвращающее объект, реализующий интерфейс **System.Security.Principal.IIdentity**. Этот объект представляет учетную запись пользователя, от имени которого исполняется код.

Интерфейс **IIdentity** определяет свойства AuthenticationType, IsAuthenticated и Name. Через них можно определить механизм

аутентификации, узнать, был ли пользователь аутентифицирован, и собственно личность пользователя. Приведем пример использования этих свойств:

```
UserLabel.Text = User.Identity.Name;
UserTypeLabel.Text = User.Identity.AuthenticationType;
UserAuthenticatedLabel.Text =
              User.Identity.IsAuthenticated.ToString();
```

Объекты, полученные через свойство User.Identity, имеют тип **System.Security.Principal.WindowsIdentity**, если использовалась аутентификация Windows, или **System.Web.Security.RolePrincipal** в случае использования аутентификации, основанной на формах.

Интерфейс **IPrincipal** также определяет метод IsInRole, позволяющий установить принадлежность к указанной группе пользователей или указанной роли в случае использования аутентификации, основанной на формах. Пример показывает, как этот метод может быть вызван:

```
if (User.IsInRole("Administrators"))
    ...
```

Список литературы

1. C# 5.0 и платформа .NET 4.5 для профессионалов : пер. с англ. / К. Нейгел, Б. Ивьен, Д. Глинн, К. Уотсон. — М. : И. Д. Вильямс, 2014. — 1440 с.

2. Шилдт, Г. C# 4.0 Полное руководство : пер. с англ. / Г. Шилдт. — М. : И. Д. Вильямс, 2011. — 1056 с.

3. Microsoft ASP.NET 4 с примерами на C# 2010 для профессионалов : пер. с англ. — 4-е изд. / М. Мак-Дональд, А. Фримен, М. Шпушта. — М. : ООО И. Д. Вильямс, 2011. — 1424 с.

4. Васильчиков, В. В. Программирование на языке C# для .NET Framework : курс лекций / В. В. Васильчиков. — Ярославль : ЯрГУ, 2013. — Часть 1. — 196 с.

5. Васильчиков, В. В. Программирование на языке C# для .NET Framework : курс лекций / В. В. Васильчиков. — Ярославль : ЯрГУ, 2014. — Часть 2. —200 с.

6. Васильчиков, В. В. Дополнительные вопросы программирования для .NET Framework : учебно-методическое пособие / В. В. Васильчиков. — Ярославль : ЯрГУ, 2017. — 60 с.

Оглавление

Введение 3
Лекция 1. Разработка Microsoft ASP.NET Web-приложений в Microsoft Visual Studio 4
Раздел 1. Введение в .NET Framework 4
Раздел 2. Обзор ASP.NET 5
Клиент-серверное взаимодействие 5
Что собой представляет ASP.NET 6
Компоненты Web-приложения ASP.NET 7
Формирование и рендеринг разметки и кода приложения 7
Динамическая модель компиляции ASP.NET 9
Расширения ASP.NET Framework 9
Лекция 2. Создание Web-приложений в Microsoft Visual Studio на языке C# 11
Раздел 1. Создание простого Web-приложения 11
Процесс разработки Web-приложения 11
Файлы и папки проекта Web-приложения для варианта Web application project 12
Файлы и папки проекта Web-приложения для варианта Web Site project 13
Сравнение вариантов Web application project и Web Site project 13
Основные файлы Web application project 15
Лекция 3. Создание ASP.NET Web-формы 17
Раздел 1. Создание Web-формы 17
Как выглядит описание Web-формы? 17
Раздел 2. Добавление серверных контролов на Web-форму и их настройка 20
Что такое серверные контролы? 20
Типы серверных контролов 21
Сохранение состояния элементов управления 26
Добавление и настройка HTML-серверных контролов и Web-серверных контролов 27
Выбор между HTML-серверными контролами и Web-серверными контролами 28
Лекция 4. Реализация функциональности ASP.NET Web-формы 30
Раздел 1. Работа с фоновым кодом 30
Варианты размещения кода 30
Использование файлов с фоновым кодом 32
Раздел 2. Обработка событий серверных контролов 33
Что такое обработчики событий? 33
Что такое обработчики клиентской стороны? 34
Что такое обработчики серверной стороны? 36
Использование свойств Web-серверных контролов 36
Раздел 3. Обработка событий уровня страницы 37
События жизненного цикла страницы 37
Процесс обратной посылки 38

Лекция 5. Создание и использование мастер-страниц и пользовательских контролов 40
Раздел 1. Мастер-страницы 40
Что такое мастер-страница? 40
Что такое контент-страница? 42
Что такое вложенная мастер-страница? 43
Как обрабатывается мастер-страница? 45
Добавление мастер-страницы к существующему Web-проекту 46
Резюме 47
Раздел 2. Пользовательские контролы 47
Варианты создания пользовательских контролов 48
Преимущества и недостатки использования пользовательских контролов 49
Преобразование Web-формы в пользовательский контрол 50
Добавление пользовательского контрола на Web-форму 51
Лекция 6. Проверка достоверности пользовательского ввода 53
Раздел 1. Обзор процесса проверки достоверности пользовательского ввода 53
Что такое валидация? 53
Проверки клиентской и серверной стороны 54
Раздел 2. Контролы проверки 55
Обзор контролов валидации 55
Основные контролы валидации и их свойства 56
Использование нескольких контролов валидации 60
Позиционирование и настройка контролов валидации на Web-форме 61
Раздел 3. Проверка Web-форм 62
Контрол ValidationSummary 62
Программная валидация Web-форм 63
Лекция 7. Средства отладки Web-приложений ASP.NET 65
Раздел 1. Отладка в ASP.NET 65
Типы ошибок 65
Что мы понимаем под отладкой? 65
Класс Debug 66
Как собирается отладочная информация 66
Методы для сбора отладочной информации 67
Локальная и удаленная отладка 68
Раздел 2. Трассировка в ASP.NET 69
Класс TraceContext 69
Трассировка Web-приложения 70
Лекция 8. Управление данными Web-приложений ASP.NET 72
Раздел 1. Обзор ADO.NET 72
Что такое ADO.NET? 72
Объектная модель ADO.NET 73
Обзор ADO.NET Entity Framework 75
Раздел 2. Подключение к базе данных 78
Создание соединения 78
Организация передачи данных 80
Раздел 3. Управление данными 83
Получение простых данных 83

Получение более сложных данных 85
Манипуляции с данными 87
Контролы для управления данными 88
Лекция 9. Использование LINQ для управления данными 90
Раздел 1. Обзор LINQ 90
Что нам дает использование LINQ? 91
Операции запроса LINQ 92
Что такое LINQ to XML? 94
Что такое LINQ to SQL? 95
Что такое LINQ to Entities? 97
Раздел 2. Управление данными с использованием LINQ to XML 98
Запрос XML-данных с использованием LINQ to XML 98
Работа с XML-данными с использованием LINQ to XML 101
Отображение данных LINQ to XML 105
Раздел 3. Управление данными с использованием LINQ to SQL и LINQ to Entities 108
Запрос SQL-данных с использованием LINQ 108
Обработка SQL-данных с использованием LINQ 111
Отображение SQL-данных с использованием LINQ 113
Лекция 10. Управление данными с использованием ASP.NET Dynamic Data 117
Раздел 1. Обзор ASP.NET Dynamic Data 117
Как работает ASP.NET Dynamic Data? 118
Устройство проекта Dynamic Data 121
Динамическое построение страниц 123
Шаблоны ASP.NET Dynamic Data 124
Маршрутизация ASP.NET Dynamic Data 126
Раздел 2. Применение ASP.NET Dynamic Data 127
Создание нового Web-сайта ASP.NET Dynamic Data 128
Добавление возможностей Dynamic Data к существующему Web-сайту 129
Добавление динамического поведения контролам для привязки данных . 131
Раздел 3. Настройка ASP.NET Dynamic Data 132
Создание шаблона страницы 132
Настройка механизма скаффолдинга 133
Лекция 11. Использование Ajax в ASP.NET 136
Раздел 1. Введение в Ajax 136
Что такое Asynchronous JavaScript and XML? 136
Как устроен ASP.NET Ajax? 137
Архитектура компонентов ASP.NET Ajax 139
Раздел 2. Добавление возможностей Ajax стандартным контролам ASP.NET 140
Основные черты Ajax ASP.NET 141
Встроенные серверные контролы ASP.NET Ajax 141
Использование элемента управления ScriptManager 142
Использование элемента управления UpdatePanel 144
Использование элемента управления UpdateProgress 147
О программным управлении частично-страничной посылкой на стороне клиента 148

Раздел 3. Элементы управления Ajax из пакета Ajax Control Toolkit ***149***
Обзор пакета Ajax Control Toolkit *150*
Обзор контролов Ajax Control Toolkit *150*
Лекция 12. Использование возможностей служб WCF **152**
Раздел 1. Введение в службы WCF ***152***
Что такое Windows Communication Foundation? *153*
Что такое служба WCF? *154*
Взаимодействие через WCF-службы *156*
Области использования WCF-служб *157*
Раздел 2. Вызов методов служб WCF из Web-формы ***158***
Использование службы Windows Communication Foundation *158*
Как добавить ссылку на службу в проект Visual Studio *159*
Обзор файлов с метаданными ссылки на службу *159*
Раздел 3. Службы данных WCF ***160***
Обзор служб данных WCF *161*
Создание службы данных WCF *162*
Лекция 13. Управление состоянием Web-приложений в ASP.NET **164**
Раздел 1. Управление данными приложения ***164***
Зачем нужно управление состоянием? *164*
Типы управления состоянием *165*
Средства управления состоянием на стороне сервера *167*
Средства управления состоянием на стороне клиента *172*
Раздел 2. Управление пользовательскими данными ***175***
Что такое профили ASP.NET? *176*
Использование профилей ASP.NET *177*
Раздел 3. Управление кэшированием ***180***
Механизмы кэширования ASP.NET *180*
Использование кэширования ASP.NET *181*
Лекция 14. Безопасность ASP.NET Web-приложений **187**
Раздел 1. Обзор вопросов безопасности ***187***
Что такое аутентификация и авторизация? *187*
Методы аутентификации ASP.NET *188*
Сценарии поддержки методов аутентификации ASP.NET *189*
Механизмы аутентификации IIS *190*
Аутентификация и авторизация, основанные на формах *193*
Протокол защищенных сокетов *193*
Раздел 2. Декларативная настройка аутентификации и авторизации ***195***
Настройка аутентификации, встроенной в Windows *196*
Настройка аутентификации, основанной на формах *197*
Настройка авторизации *197*
Раздел 3. Программная настройка аутентификации и авторизации ***199***
Обзор Login-контролов *199*
Доступ к данным, идентифицирующим пользователя *201*
Список литературы **203**
Оглавление **204**

Учебное издание

Васильчиков Владимир Васильевич

Программирование ASP.NET Web Forms

Учебно-методическое пособие

Редактор, корректор Л. Н. Селиванова
Верстка В. В. Васильчиков

Подписано в печать 20.02.2021 Формат 60×84 1/16.
Усл. печ. л. 12,09. Уч.-изд. л. 8,1.
Тираж 3 экз. Заказ

Оригинал-макет подготовлен
в редакционно-издательском отделе ЯрГУ.

Ярославский государственный университет
им. П. Г. Демидова.
150003, Ярославль, ул. Советская, 14.