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

Кафедра информатики

А.А. Волосевич

ЯЗЫК C# И ПЛАТФОРМА .NET
(часть 6)

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

Минск 2009

СОДЕРЖАНИЕ

6. ВЕБ-СЛУЖБЫ, WCF И AJAX

6.1. ВЕБ-СЛУЖБЫ

В процессе эволюции Интернета появилась необходимость в создании *распределённых приложений*, то есть приложений, компоненты которых размещены на различных компьютерах и взаимодействуют через локальную или глобальную сеть. *Веб-службы* (или *веб-сервисы*) – это один из способов создания распределённых приложений. Протоколы веб-служб стандартизированы, а, значит, не зависят от языка и платформы реализации.

Начнём с некоторых терминов, используемых при работе с веб-службами:

1. *SOAP* – протокол для взаимодействия с веб-службой, основанный на XML[1]. Сообщение SOAP - это обычный XML-документ особой схемы.
2. *WSDL* (*Web Services Description Language*) - язык описания веб-служб, основанный на XML. WSDL-документ используется клиентами службы, чтобы узнать, какие методы предлагает служба, каков тип и число параметров этих методов.
3. *DISCO* (сокр. *Discovery*) и *UDDI* (*Universal Description, Discovery, and Integration*) - стандарты для поиска и публикации веб-служб. При использовании DISCO на сайте размещается файл, который содержит список имеющихся веб-служб. UDDI - это централизованный каталог, в котором веб-службы опубликованы по именам компаний. Это также место, куда потенциальные клиенты могут обратиться, чтобы отыскать веб-службу. Сама работа с UDDI основана на использовании веб-служб.

Рассмотрим создание простой веб-службы на примере службы `EmployeeService`, предоставляющей информацию о сотрудниках некой компании. В платформе .NET веб-службы – часть инфраструктуры ASP.NET, или, иначе говоря, веб-службы являются элементами веб-приложений. Для каждой веб-службы необходимо создать класс с методами службы и связанную с классом *точку входа* – файл с расширением `asmx`. Каждый `asmx`-файл начинается с директивы `@WebService`, которая должна объявить серверный язык и класс веб-службы. Директива `@WebService` может задавать и другую информацию, например, имя файла Code Behind. Приведём пример файла `EmployeeService.asmx`:

```
<%@ WebService Language="C#" Class="EmployeeService"
                CodeBehind="~/App_Code/EmployeeService.cs" %>
```

Как видно из примера, используется файл Code Behind, размещенный в папке `App_Code`. Ниже приведён код класса `EmployeeService`.

```
using System.Data;
using System.Data.SqlClient;
using System.Web.Services;
```

[1] Ранее SOAP было аббревиатурой: *Simple Object Access Protocol*.

```
public class EmployeeService
{
    const string ConnectionString = ". . .";
    const string GetCount = "SELECT COUNT(*) FROM Employees";
    const string GetAll = "SELECT * FROM Employees";

    [WebMethod]
    public int GetEmployeesCount()
    {
        var connection = new SqlConnection(ConnectionString);
        var command = new SqlCommand(GetCount, connection);
        connection.Open();
        var numEmployees = (int) command.ExecuteScalar();
        connection.Close();
        return numEmployees;
    }

    [WebMethod]
    public DataSet GetEmployeesData()
    {
        var connection = new SqlConnection(ConnectionString);
        var dataAdapter = new SqlDataAdapter(GetAll, connection);
        var dataSet = new DataSet();
        dataAdapter.Fill(dataSet, "Employees");
        return dataSet;
    }
}
```

Обратите внимание, что класс веб-службы не содержит полей, то есть, *не поддерживает состояние*. Дело в том, что объекты классов веб-служб, как и страницы ASP.NET, создаются для каждого вызова клиента. Методы класса веб-службы, доступные клиенту, помечаются атрибутом `[WebMethod]`.

Существует ограничение на тип входных и выходных параметров методов веб-службы. Это ограничение диктуется тем фактом, что веб-службы используют стандарты обмена, основанные на XML. Набор типов данных для веб-служб ограничивается типами, представимыми в XML Schema (см. табл. 1).

Таблица 1

Типы данных, поддерживаемые веб-службами

Тип данных	Описание
Примитивные типы	Примитивные типы данных платформы .NET (числовые типы, булевский, строковый, символьный, байтовый) и тип даты-времени
Перечисления	Типы перечислений (`enum`) поддерживаются. Веб-служба использует строковое имя значения перечисления
`XmlNode`	Объекты этого типа представляют собой часть XML-документа. Их можно использовать для отправки произвольных XML-данных
`DataSet` и `DataTable`	Объекты этих типов можно применить для возврата информации из реляционной базы данных. Другие объекты ADO.NET (например, `DataColumn` или `DataRow`) не поддерживаются

Массивы	Можно использовать массивы любого поддерживаемого типа (байтовые массивы автоматически кодируются в Base64). Поддерживаются универсальные списковые типы (`List<T>`), которые просто преобразуется в массив. Коллекции-словари не поддерживаются
Пользовательские классы и структуры	Можно передавать любой объект, который создан на основе пользовательского класса или структуры. Передаются только `public`-элементы. Их тип – один из поддерживаемых веб-службами

ASP.NET позволяет протестировать веб-службу. Для этого достаточно запустить приложение и обратиться к концевой точке службы в браузере (рис. 1).

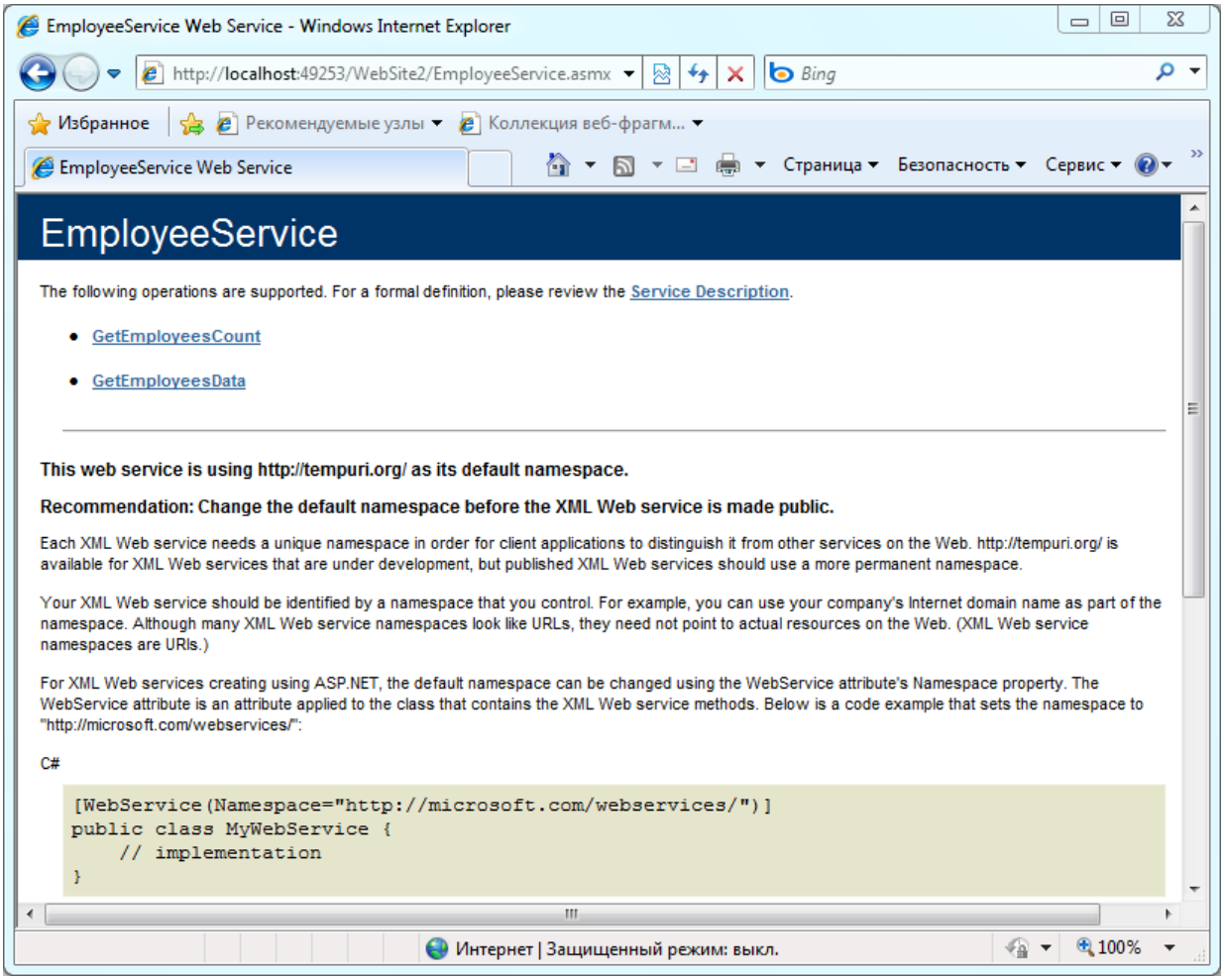

Рис. 1. Тестирование веб-службы через браузер.

Чтобы использовать веб-службу в коде клиента, необходим специальный *прокси-класс*. Прокси-класс обладает методами, сигнатура которых идентична методам класса веб-службы. Методы прокси-класса выполняют сетевое соединение с сервером веб-службы, формирование правильных SOAP-пакетов для передачи серверу, интерпретацию данных, принятых от веб-службы.

Существуют два основных способа создания прокси-класса:

- Применение утилиты командной строки `wsdl.exe`.
- Использование сервисных ссылок в Visual Studio.

Для генерации прокси-класса с применением `wsdl.exe` требуется указать адрес службы в качестве параметра этой утилиты:

```
wsdl http://localhost:49253/WebSite/EmployeeService.asmx
```

Дополнительные ключи утилиты `wsdl.exe` позволяют указать язык программирования, на котором генерируется прокси-класс, имя класса и его пространство имен и прочее. В Visual Studio 2008 прокси-класс можно построить, использовав в дереве проекта команду `References|Add Service Reference`. Сгенерированный прокси-класс будет создан по правилам технологии Windows Communication Foundation.

После описания базовых приёмов создания и использования веб-служб, коснёмся вопросов их углубленной настройки. Класс веб-службы может (но не обязательно) наследоваться от класса `System.WebServices.WebService`. Наследование позволяет получить доступ к встроенным объектам ASP.NET - `Application`, `Context`, `Server`, `Session`, `Site`, `User` - посредством соответствующих свойств. Впрочем, для доступа к объектам ASP.NET можно применять и `HttpContext.Current`. К классу веб-службы можно применить атрибут `[WebService]`. Свойства этого атрибута позволяют указать описание и имя службы, а также пространство имен XML, используемое службой.

Свойства атрибута `[WebMethod]` позволяют дополнительно настроить метод службы. Эти свойства описаны в табл. 2.

Таблица 2

Свойства атрибута `[WebMethod]`

Имя свойства	Описание
`BufferResponse`	Буферизация ответа веб-метода (улучшает производительность). По умолчанию буферизация включена (значение `true`)
`CacheDuration`	Количество секунд, на которые результат метода записывается в кеш. По умолчанию – 0 (кеш не используется)
`Description`	Строковое описание метода
`EnableSession`	Поддержка сессий в классе веб-службы. По умолчанию значение этого свойства `false` – поддержка сессий выключена
`MessageName`	Имя метода, используемое клиентом службы. Свойство часто используется, если в классе веб-службы методы перегружены (тогда им надо задать разные имена)
`TransactionOption`	Свойство управляет поддержкой транзакций в веб-службе. Значением свойства являются элементы перечисления `TransactionOption`. По умолчанию поддержка транзакций отключена

Затрагивая аспекты работы клиента с прокси-классом службы, отметим, что прокси-класс позволяет вызывать методы службы асинхронно (соответствующие методы завершаются суффиксом `Async`). Чтобы обработать ответ веб-службы в случае асинхронного вызова, применяется техника назначения обработчика события, генерируемого при завершении вызова.

6.2. ОСНОВЫ WINDOWS COMMUNICATION FOUNDATION

В данном параграфе рассматриваются базовые термины и понятия технологии Windows Communication Foundation, представленной в .NET Framework 3.0. *Windows Communication Foundation* (далее *WCF*) - единая программная модель, предназначенная для создания распределённых и сервис-ориентированных приложений. В свою очередь, *сервис-ориентированная архитектура* (*service-oriented architecture*, *SOA*) – это модульный подход к разработке программного обеспечения, основанный на использовании сервисов (служб) со стандартизированными интерфейсами. WCF позволяет разработчикам строить безопасные, надежные решения, которые могут взаимодействовать с различными платформами и существующими сервисами.

В основе функционирования WCF лежат так называемые *конечные точки* (*endpoints*), состоящие из адреса, привязки и контракта (*address-binding-contract, ABC*). *Адрес* задает местоположение конечной точки. *Привязка* определяет протокол, на основе которого будет происходить взаимодействие. *Контракт* регламентирует, какие операции умеет выполнять сервис.

Любой адрес, используемый в WCF, имеет формат:

```
[базовый адрес]/[опциональный URI]
```

В свою очередь, базовый адрес состоит из следующих частей:

```
[транспорт]://[машина или домен][:порт(опционально)]
```

Здесь `[транспорт]` указывает на протокол коммуникации:

- `net.tcp` – для протокола TCP;
- `http` (или `https`) – для протокола HTTP;
- `net.pipe` – при использовании именованных каналов.

Для протокола TCP в адресе обычно указывается и порт. Если этого не сделать, используется порт 808. Для протокола HTTP по умолчанию используется порт 80. Адреса TCP и HTTP позволяют *разделять порт*, то есть такие адреса могут различаться только указанием опционального URI. Если в качестве транспорта используются именованные каналы, то адрес имеет форму `net.pipe://localhost/MyPipe`. Необходимо указывать или имя машины, или `localhost` (коммуникация между машинами запрещена). Последняя часть адреса рассматривается как имя канала.

Как было указано выше, в WCF привязка определяет транспортный протокол для взаимодействия клиента и службы[1]. Кроме этого, привязка задаёт формат передаваемых данных. От выбора привязки зависит поддержка транзакций, сессий, возможность дуплексного обмена. Можно использовать одну из стандартных привязок, или, при необходимости, создать собственную привязку.

[1] Обратите внимание, что информацию о протоколе содержит и адрес. Следовательно, выбор привязки отчасти диктует формат используемого адреса.

Таблица 3

Основные привязки WCF

Имя привязки	Описание	Транспорт
`basicHttpBinding`	Привязка для веб-служб, совместимых с WS-I Basic Profile 1.1 (в частности, asmx-службы)	`HTTP/HTTPS`
`netTcpBinding`	Привязка для коммуникации между двумя .NET приложениями по протоколу TCP	`TCP`
`netPeerTcpBinding`	Привязка для построения пиринговых сетей	`P2P`
`netNamedPipeBinding`	Привязка, использующая именованные каналы. Применяется для взаимодействия двух .NET приложений на **одной** машине	`IPC`
`wsHttpBinding`	Привязка для веб-служб с дополнительными возможностями (безопасность, транзакции)	`HTTP/HTTPS`
`wsFederationHttpBinding`	Привязка для продвинутых веб-служб с интегрированной идентификацией	`HTTP/HTTPS`
`wsDualHttpBinding`	Аналог `wsHttpBinding`, но поддерживает двухсторонние коммуникации на основе дуплексных контрактов	`HTTP`
`netMsmqBinding`	Привязка, использующая в качестве транспорта Microsoft Message Queue	`MSMQ`
`msmqIntegrationBinding`	Аналог `netMsmqBinding`, но для работы с унаследованным ПО на основе MSMQ	`MSMQ`

Любые службы WCF реализуют сервисные контракты. *Сервисный контракт* – это платформенно-независимый стандартизированный способ описания того, что может делать служба. Кроме этого, на основе контракта на клиенте строится прокси-класс для взаимодействия со службой[1].

Если в случае веб-служб их средой выполнения (или *хостингом*) было веб-приложение ASP.NET, то для хостинга WCF-служб может использоваться несколько вариантов:

1. WCF-служба входит в состав сайта ASP.NET (подобно веб-службам).
2. Хостингом WCF-службы является сервер IIS (это, по сути, ASP.NET-сайт, состоящий из одной WCF-службы).
3. WCF-служба может размещаться в Windows-приложении .NET (включая системные сервисы).

Клиентские WCF-приложения применяют для работы со службами прокси-классы. Работа прокси-классов основывается на использовании каналов. *Канал* – это клиентский объект, агрегирующий информацию о сервисном контракте и одной концевой точке вызываемой службы.

[1] Кроме сервисных контрактов, в WCF существуют *контракты данных* (определяют типы данных, которые можно передавать между клиентом и службой) и *контракты исключений* (описывают, какие исключения может генерировать служба, и как информация об этом передается клиенту).

6.3. ДЕМОНСТРАЦИОННОЕ WCF-ПРИЛОЖЕНИЕ

В этом параграфе рассматриваются основные шаги, выполняемые для создания простой WCF-службы и клиентского приложения. Наша служба будет содержать два метода. Метод `GetCustomer()` возвращает строку с информаций о покупателе по идентификатору покупателя. Метод `GetCustomersCount()` возвращает общее число покупателей.

1. Создание сервисного контракта.

Начнём разработку службы с создания сервисного контракта. На платформе .NET сервисным контрактом становится класс или интерфейс, помеченный атрибутом `[ServiceContract]`[1]. Обычно атрибут `[ServiceContract]` применяется к интерфейсу – это позволяет разделить описание контракта между службой и клиентом. Если же этот атрибут применяется к классу, то класс должен иметь `public`-конструктор без параметров. Методы сервисного контракта, доступные клиенту, должны быть помечены атрибутом `[OperationContract]`. Помечены могут быть только методы – свойства или поля пометить нельзя.

```
using System.ServiceModel;

namespace ServiceInterface
{
    [ServiceContract]
    public interface ICustomer
    {
        [OperationContract]
        string GetCustomer(int customerId);

        [OperationContract]
        int GetCustomersCount();
    }
}
```

Сервисный контракт разместим в сборке `ServiceInterface.dll`. В дальнейшем эта сборка будет использоваться как сервером, так и клиентом.

2. Реализация сервисного контракта.

Следующим шагом будет создание класса, реализующего сервисный контракт на стороне сервера. Опишем класс `Customer` в консольном приложении. В дальнейшем это консольное приложение исполнит роль хостинга WCF-службы.

```
using ServiceInterface;
namespace Host
{
    public class Customer : ICustomer
    {
```

[1] Для работы с WCF вы должны добавить к проекту ссылку на сборку `System.ServiceModel` и использовать одноимённое пространство имен.

```
        public string GetCustomer(int customerId)
        {
            return "Customer " + customerId;
        }

        public int GetCustomersCount()
        {
            return 10;
        }
    }

    public class Program
    {
        private static void Main() { }
    }
}
```

3. Хостинг WCF-службы.

Организуем хостинг WCF-службы. Для этого нужно описать концевую точку службы (или несколько концевых точек) и запустить службу.

Описание концевой точки выполняется или программно, или декларативно (в файле конфигурации). При программном подходе задаётся адрес (можно использовать класс `System.Uri` или строку), объект класса-привязки, и используется класс `ServiceHost` для связывания всех элементов и запуска службы:

```
// Реализация метода Main() из предыдущего фрагмента кода
private static void Main()
{
    // 1. Создаем объект, описывающий адрес службы
    var uri = new Uri("net.tcp://localhost:8228/Customer");

    // 2. Создаем объект, описывающий привязку
    var binding = new NetTcpBinding();

    // 3. Создаём хостинг для службы
    var host = new ServiceHost(typeof(Customer));
    host.AddServiceEndpoint(typeof(ICustomer), binding, uri);

    // 4. Запускаем хостинг
    host.Open();
    Console.ReadLine();
}
```

Если применяется декларативная настройка концевой точки, то используется секция конфигурационного файла `<system.serviceModel>`. В подсекцию `<services>` помещается описание служб. Каждая служба характеризуется име-

нем и описанием концевой точкой (или точек). В свою очередь, концевая точка содержит указание на адрес, привязку и контракт[1].

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name="Host.Customer">
        <endpoint address="net.tcp://localhost:8228/Customer"
                  binding="netTcpBinding"
                  contract="ServiceInterface.ICustomer" />
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

При декларативной настройке концевой точки достаточно создать в программе объект класса `ServiceHost` и запустить хостинг.

```
private static void Main()
{
    var host = new ServiceHost(typeof(Customer));
    host.Open();
    Console.ReadLine();
}
```

4. Построение клиентского приложения.

Перейдем к созданию клиентского приложения, использующего WCF-службу. В пространстве имён `System.ServiceModel` содержится абстрактный универсальный класс `ClientBase<T>`, который можно использовать как базовый класс для клиентского прокси. Параметр `T` задаёт тип клиентского канала.

```
public class CustomerClient : ClientBase<ICustomer>, ICustomer
{
    public CustomerClient() { }

    public CustomerClient(string endpointConfigurationName) :
            base(endpointConfigurationName) { }

    public CustomerClient(string endpointConfigurationName,
                          string remoteAddress) :
            base(endpointConfigurationName, remoteAddress) { }

    public CustomerClient(string endpointConfigurationName,
                          EndpointAddress remoteAddress) :
            base(endpointConfigurationName, remoteAddress) { }
```

[1] В .NET Framework SDK существует утилита `SvcConfigEditor.exe`, облегчающая создание конфигурационных секций для WCF-служб, как на стороне сервера, так и на клиенте.

```
    public CustomerClient(
                        System.ServiceModel.Channels.Binding binding,
                           EndpointAddress remoteAddress) :
                base(binding, remoteAddress) { }

    public string GetCustomer(int customerId)
    {
        return base.Channel.GetCustomer(customerId);
    }

    public int GetCustomersCount()
    {
        return base.Channel.GetCustomersCount();
    }
}
```

Обратите внимание на следующие аспекты. Чтобы определить тип канала, указан интерфейс `ICustomer` - сервисный контракт. Очевидно, что в этом примере предполагается доступность для клиента сборки `ServiceInterface.dll`. Для удобства настройки канала класс `CustomerClient` определяет несколько конструкторов, просто вызывающих конструкторы базового класса. Класс `CustomerClient` реализует интерфейс `ICustomer`. В соответствующих методах вызовы делегируются внутреннему объекту `Channel`.

Объект прокси-класса будет нуждаться в дополнительной настройке перед использованием. Как и в случае WCF-службы, настройку можно выполнить или программно, или в файле конфигурации.

В следующем примере кода настройка выполняется программно:

```
private static void Main(string[] args)
{
    // 1. Создаем объект, описывающий адрес концевой точки
    var address =
           new EndpointAddress("net.tcp://localhost:8228/Customer");

    // 2. Создаем объект, описывающий привязку
    var binding = new NetTcpBinding();

    // 3. Создаем прокси-класс
    var client = new CustomerClient(binding, address);

    // 4. Работаем с сервисом через прокси
    Console.WriteLine(client.GetCustomer(23););
}
```

Если настройка выполняется в конфигурационном файле, то указываются адрес, привязка, контракт и, дополнительно, имя настраиваемой точки:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
```

```
  <system.serviceModel>
    <client>
      <endpoint name="Customer"
                address="net.tcp://localhost:8228/Customer"
                binding="netTcpBinding"
                contract="ServiceInterface.ICustomer" />
    </client>
  </system.serviceModel>
</configuration>
```

Имя точки используется при создании объекта прокси-класса.

```
var client = new CustomerClient("Customer");
```

Заметим, что обычно прокси-классы не создаются вручную - используется специальная утилита `svcutil.exe`. Данная утилита может создать прокси-класс, даже если клиенту не известен сервисный контракт. Правда, для этого WCF-служба должна обладать особой дополнительной концевой точкой, описывающей её *метаданные* (концевая точка MEX).

6.4. ДОПОЛНИТЕЛЬНЫЕ АСПЕКТЫ ИСПОЛЬЗОВАНИЯ WCF

1. Хостинг WCF-служб в приложениях ASP.NET.

Чтобы поместить WCF-службу в веб-приложение ASP.NET, добавьте в состав сайта элемент `WCF service`. Сделав это для службы с именем `Service`, мы получим файлы `Service.svc`, `Service.svc.cs` и `IService.cs`. Последний файл – это сервисный контракт. Содержимое первых двух файлов представлено ниже.

Service.svc:

```
<%@ ServiceHost Language="C#" Debug="true" Service="WebApp.Service"
                CodeBehind="Service.svc.cs" %>
```

Service.svc.cs:

```
namespace WebApp
{
    public class Service : IService
    {
        public void DoWork() { }
    }
}
```

Два файла просто реализуют подход Code Behind. При необходимости программный код можно разместить непосредственно в файле `Service.svc` сразу после директивы `@ServiceHost`.

Как и для веб-служб, можно тестировать WCF-службу из браузера. Для создания клиентского прокси используется команда Visual Studio `Add Service Reference` или утилита `svcutil.exe` в форме:

```
svcutil.exe http://localhost:2349/Service.svc?wsdl
```

2. Контракты.

Атрибуты [ServiceContract] и [OperationContract] обладают набором свойств, позволяющих более «тонко» настроить сервисный контракт.

Таблица 4

Свойства атрибута [ServiceContract]

Имя	Описание
ConfigurationName	Определяет имя службы в конфигурационном файле. По умолчанию используется полное имя класса, реализующего службу
CallbackContract	Когда служба используется для дуплексного обмена сообщениями, это свойство определяет контракт, реализованный клиентом
Name и Namespace	Задают имя и пространство имён для элемента <portType> в документе WSDL. Namespace рекомендуется указывать
SessionMode	Позволяет организовать в WCF-службе поддержку состояния сеанса. Возможные значения: Allowed, NotAllowed и Required; все они определены в перечислении SessionMode
ProtectionLevel	Определяет, должна ли привязка поддерживать защиту коммуникаций. Возможные значения определены перечислением ProtectionLevel: None, Sign, EncryptAndSign

Таблица 5

Свойства атрибута [OperationContract]

Имя	Описание
Action	WCF использует Action из запроса SOAP для отображения его на соответствующий метод. По умолчанию это комбинация из пространства имен XML контракта, имени контракта и имени операции. Вы можете переопределить значение Action, специфицируя свойство Action
ReplyAction	Устанавливает SOAP-имя Action ответного сообщения
AsyncPattern	Если операция реализуется с использованием асинхронного шаблона, установите свойство в true
IsInitiating IsTerminating	Если контракт состоит из последовательности операций, то инициирующей операции назначается свойство IsInitiating, а последней операции последовательности требуется свойство IsTerminating. Инициирующая операция запускает новый сеанс; а в операции завершения сервер закрывает сеанс
IsOneWay	Если IsOneWay = true, клиент не ожидает ответного сообщения. Инициатор однонаправленного сообщения не имеет прямого способа обнаружить сбой после отправки своего запроса
Name	Имя операции по умолчанию - это имя метода, которому назначена операция контрактом. Вы можете изменить имя операции, применив свойство Name
ProtectionLevel	С помощью свойства ProtectionLevel вы определяете, должно ли сообщение быть снабжено подписью или зашифровано

Контракт данных – это класс или структура со специальными атрибутами, устанавливающими соответствие между элементами типа и элементами XML документа. Контракты данных следует применять в том случае, если методы WCF-службы получают или возвращают сложные типы данных.

Чтобы превратить тип в контракт данных, требуется применить к типу атрибут [DataContract], а к полям или свойствам, подлежащим сериализации, – атрибут [DataMember][1]. Видимость элементов при этом роли не играет. Подробнее о контрактах данных рассказано в параграфе, посвящённом сериализации.

3. Поведения.

Класс, реализующий WCF-службу, может быть помечен атрибутом [ServiceBehavior]. Этот атрибут используется для описания *поведения* службы.

Важные свойства поведения - InstanceContextMode и ConcurrencyMode. Их значениями являются элементы одноимённых перечислений. Первое свойство управляет режимом создания объектов службы. Второе – режимом параллелизма при обработке запросов клиентов. Сочетания значений свойств описаны в табл. 6.

Таблица 6

Сочетание поведений InstanceContextMode и ConcurrencyMode

<table>
<tr><td colspan="2" rowspan="2"></td><td colspan="3">InstanceContextMode</td></tr>
<tr><td>Single</td><td>PerCall</td><td>PerSession</td></tr>
<tr><td rowspan="3">ConcurrencyMode</td><td>Single
(по умолч.)</td><td>Для обработки всех запросов создаётся один объект и один поток. Пока текущий запрос обрабатывается, следующие ставятся в очередь</td><td rowspan="3">При каждом обращении создается новый объект. Режим параллелизма не имеет значения</td><td>На каждый сеанс связи с клиентом создается один объект и один поток. Асинхронные запросы клиента ставятся в очередь</td></tr>
<tr><td>Reentrant</td><td>Для обработки всех запросов создаётся один объект и один поток. Пока текущий запрос обрабатывается, следующие ставятся в очередь</td><td>На каждый сеанс связи с клиентом создается один объект и один поток. Этот поток может покинуть метод, сделать что-то, а потом вернутся (асинхронное кодирование на сервере)</td></tr>
<tr><td>Multiple</td><td>Создается один объект, но с ним могут параллельно работать несколько потоков (клиентов). Нужно позаботиться о защите членов класса с помощью механизмов синхронизации</td><td>На каждый сеанс связи с клиентом создается один объект, с которым могут параллельно работать несколько потоков (асинхронные операции). Нужно позаботиться о защите элементов (синхронизация)</td></tr>
</table>

Для привязок с поддержкой сессий по умолчанию используется поведение InstanceContextMode.PerSession, для привязок без поддержки сессий - PerCall. Сессии поддерживают привязки wsHttpBinding, wsDualHttpBinding, netTcpBinding, netNamedPipeBinding. Кроме этого, поддержка сессий должна быть декларирована в сервисном контракте (свойство SessionMode).

[1] Данные атрибуты размещаются в пространстве имён System.Runtime.Serialization и одноимённой сборке.

6.5. ОСНОВЫ ТЕХНОЛОГИИ AJAX

Сокращение AJAX происходит от *Asynchronous JavaScript and XML* (*асинхронный код JavaScript и XML*). Этим общим термином обозначаются высокоинтерактивные веб-приложения, быстро реагирующие на действия пользователя. Термин AJAX появился в 2004 году в Java-сообществе. Первоначально он использовался для обозначения семейства взаимосвязанных технологий, реализующих различные формы удаленного исполнения сценариев. В наши дни все разновидности удаленных сценарных технологий обычно помечаются аббревиатурой AJAX.

Основой удаленного исполнения сценариев является возможность отправки асинхронных HTTP-запросов. В данном контексте под *асинхронным вызовом* понимается запрос HTTP, который выдается за пределами встроенного модуля, обеспечивающего отправку форм HTTP. Асинхронный вызов инициируется неким событием HTML-страницы и обслуживается компонентом-посредником. В новейших AJAX-решениях таким посредником является объект `XMLHttpRequest`. Компонент-посредник отправляет обычный запрос HTTP и дожидается - синхронно или асинхронно - завершения его обработки. Получив готовые данные ответа, посредник вызывает JavaScript-функцию обратного вызова. Эта функция должна обновить все части страницы, нуждающиеся в обновлении.

Исторически первая реализация `XMLHttpRequest` появилась в Internet Explorer 5 в виде объекта ActiveX. Затем `XMLHttpRequest` был реализован в других браузерах, но так как ActiveX поддерживается только в Internet Explorer, другие браузеры обычно реализуют `XMLHttpRequest` в виде внутреннего объекта. Эти факты накладывают свой отпечаток на клиентской процедуре создания объекта `XMLHttpRequest`. Одна из реализаций этой процедуры выглядит следующим образом (код на языке JavaScript):

```
function getXMLHTTP() {
    var XMLHTTP = null;
    if (window.ActiveXObject) {
       try {
           XMLHTTP = new ActiveXObject("Msxml2.XMLHTTP");
       }
       catch (e) {
           try {
               XMLHTTP = new ActiveXObject("Microsoft.XMLHTTP");
           }
           catch (e) { }
       }
    }
    else if (window.XMLHttpRequest) {
            try {
                XMLHTTP = new XMLHttpRequest();
            }
```

```
                catch (e) { }
            }
        return XMLHTTP;
    }
```

Любая реализация объекта `XMLHttpRequest` поддерживает стандартный набор свойств и методов. В табл. 7 описаны свойства `XMLHttpRequest`.

Таблица 7

Стандартный набор свойств объекта `XMLHttpRequest`

Свойство	Описание
`readyState`	Числовой код - текущее состояние запроса. Возможные значения: целые числа от 0 до 4. 0 - «Запрос не инициализирован», 1 - «Метод `open()` вызван успешно», 2 - «Метод `send()` вызван успешно», 3 - «Прием данных», 4 - «Ответ получен»
`onreadystatechange`	Функция обратного вызова (событие), вызываемая при изменении `readyState`
`status`	HTTP-статус ответа (например, 200 - *OK*, 400 - *Ресурс не найден*)
`statusText`	Текстовое описание HTTP-статуса
`responseText`	Ответ сервера в виде текста
`responseXML`	Ответ сервера в виде XML-объекта

Из методов объекта `XMLHttpRequest` наиболее часто применяются два. Метод `open()` служит для подготовки запроса. Этот метод может принимать до пяти параметров, но обычно используются первых два: тип запроса (обычно `"GET"` или `"POST"`) и целевой URI. Третий параметр по умолчанию установлен в `true`, что означает асинхронное поведение (если запрос нужно выполнить синхронно, установите третий параметр в `false`). Метод `send()` посылает подготовленный запрос на сервер.

Работая с `XMLHttpRequest`, программист обычно выполняет следующие четыре задачи:

1. Создает объект `XMLHttpRequest`, как показано выше.
2. Устанавливает ссылку на функцию в свойстве `onreadystatechange`.
3. Вызывает у объекта `XMLHttpRequest` метод `open()` (подготовка запроса).
4. Посылает запрос, используя метод `send()`.

```
var XMLHTTP = getXMLHTTP();
if (XMLHTTP != null) {
    XMLHTTP.open("GET", "ajax.aspx?sendData=ok");
    XMLHTTP.onreadystatechange = stateChanged;
    XMLHTTP.send(null);
}
function stateChanged()
{
    if (XMLHTTP.readyState == 4 && XMLHTTP.status == 200) {
        window.alert(XMLHTTP.responseText);
    }
}
```

Естественно, что серверный код должен ожидать асинхронный запрос и правильно обработать его. Следующий код демонстрирует метод `Page_Load()`, который можно использовать как серверный обработчик для предыдущего примера.

```
protected void Page_Load()
{
    if (Request.QueryString["sendData"] != null &&
        Request.QueryString["sendData"] == "ok")
    {
        Response.Write("Hello from the server!");
        Response.End();
    }
}
```

Предыдущий пример клиентского кода оправлял на сервер GET-запрос. Если планируется отправлять информацию в POST-запросе, нужно установить правильный параметр метода `open()`. Сами данные передаются в виде набора пар как параметр метода `send()`.

```
XMLHTTP.open("POST", "ajax.aspx");
XMLHTTP.onreadystatechange = stateChanged;
XMLHTTP.send("sendData=ok&returnValue=123");
```

В некоторых ситуациях (например, посылка SOAP-запроса веб-службам) методу `send()` передается XML-строка. Однако при этом обычно необходимо указать тип контента:

```
XMLHTTP.open("POST", "ajax.aspx");
XMLHTTP.onreadystatechange = stateChanged;
XMLHTTP.setRequestHeader("Content-Type", "text/xml");
XMLHTTP.send("<soap:Envelope>...</soap:Envelope>");
```

Свойство `responseXML` представляет ответ сервера в виде объекта `XMLDocument`. Предположим, что метод `Page_Load()` из предыдущего примера модифицирован следующим образом (обратите внимание на установку свойства `Response.ContentType`):

```
protected void Page_Load()
{
    if (Request.QueryString["sendData"] != null &&
        Request.QueryString["sendData"] == "ok")
    {
        string xml = "<book title='Programming ASP.NET AJAX'>
                      <chapters>
                        <chapter number='1' title='Introduction' />
                        <chapter number='2' title='JavaScript' />
                      </chapters></book>";
        Response.ContentType = "text/xml";
```

```
            Response.Write(xml);
            Response.End();
        }
    }
```

Следующий JavaScript-код выполняет парсинг XML-информации и формирует содержимое HTML-элемента с тегом "output"[1]:

```
var xml = XMLHTTP.responseXML;
var root = xml.documentElement;
document.getElementById("output").innerHTML =
                                          root.getAttribute("title");

var list = document.getElementById("list");
var chapters = xml.getElementsByTagName("chapter");
for (var i=0; i<chapters.length; i++)
{
    var listItem = document.createElement("li");
    var listItemText = document.createTextNode(
                       chapters[i].getAttribute("number") + ": " +
                       chapters[i].getAttribute("title"));
    listItem.appendChild(listItemText);
    list.appendChild(listItem);
}
```

Несмотря на популярность формат XML, многие программисты при работе с AJAX используют формат JSON. *JavaScript Object Notation* – текстовый формат обмена данными. Достоинствами JSON являются простота, компактность и легкая интеграция с JavaScript.

JSON строится на двух структурах:

- *Набор пар «имя и значение».* В различных языках это реализовано как объект, запись, структура, словарь, хэш-таблица, список с ключом или ассоциативный массив.
- *Пронумерованный набор значений.* Во многих языках это реализовано как массив, вектор, список или последовательность.

Указанные универсальные структуры данных теоретически поддерживаются (в той или иной форме) всеми современными языками программирования. В JSON используются следующие форматы структур. *Объект* - неупорядоченное множество пар «имя и значение», заключенное в фигурные скобки. Имя отделяется от значения символом ":", а пары разделяются запятыми. *Массив* (одномерный) - множество значений, имеющих порядковые номера (индексы). Массив заключается в квадратные скобки, значения отделяются запятыми. И в объекте, и в массиве значение может быть строкой в двойных кавычках, чис-

[1] В Internet Explorer манипуляции со структурой документа можно производить только после полной загрузки документа. В этом случае можно код отправки асинхронного события записать в обработчик window.onload.

лом, `true` или `false`, `null`, объектом, массивом. Структуры *объект* и *массив* могут быть вложены друг в друга.

Ниже приведён пример информации, записанной в формате JSON:

```
{"book": {
    "title": "Programming ASP.NET AJAX",
    "author": "Christian Wenz",
    "chapters": [ {"number": "1", "title": "Introduction"},
                  {"number": "2", "title": "JavaScript"} ]
    }
}
```

Легкость работы с JSON в языке JavaScript иллюстрируется следующим примером. Обратите внимание на вызов функции `eval()`, которая порождает объект по его JSON-описанию.

```
var json = ...;
var obj = eval("(" + json + ")");
for (var i=0; i < obj.book.chapters.length; i++)
{
    document.write("<p>" + obj.book.chapters[i].number + ": " +
                   obj.book.chapters[i].title +
                   "</p>");
}
```

Начиная с версии 3.5, в платформе .NET имеется поддержка JSON-сериализации. Это разновидность сериализации контрактов данных.

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Json;

[DataContract]
public class Person
{
    [DataMember]
    public string Name { get; set; }

    [DataMember]
    public int Age { get; set; }
}

public class Program
{
    private static void Main()
    {
        var person = new Person {Name = "John", Age = 42};
        var stream = new MemoryStream();
```

```
            var serializer =
                        new DataContractJsonSerializer(typeof(Person));
            serializer.WriteObject(stream, person);
            stream.Position = 0;
            Console.Write("JSON form of Person: ");
            Console.WriteLine(new StreamReader(stream).ReadToEnd());
            stream.Position = 0;
            person = (Person)serializer.ReadObject(stream);
            Console.WriteLine("Name {0}: Age {1}",
                              person.Name, person.Age);
        }
    }
```

6.6. ПОДДЕРЖКА AJAX В ASP.NET

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

1. Элемент управления `ScriptManager`.

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

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

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

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

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

namespace WebProject
```

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

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

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

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

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

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

</script>
```

2. Элемент управления `UpdatePanel`.

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

Приведем пример фрагмента страницы, использующей `UpdatePanel`:

```
<form id="form1" runat="server">
  <div>
```

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

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

Основные свойства элемента `UpdatePanel` приведены в таблице 8.

Таблица 8

Свойства элемента `UpdatePanel`

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

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

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

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

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

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

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

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

6.7. СОЗДАНИЕ RESTFUL СЛУЖБ В WCF

REST – это архитектурный стиль, применяемый при построении распределённых приложений[1]. REST основывается на следующих принципах.

- *Клиенты* используют *глобальную сеть* для доступа к *ресурсам*. Ресурсом может быть всё, что можно поименовать и представить (файл, изображение, программа для генерации ресурса).
- Каждый ресурс имеет уникальный идентификатор (*Uniform Resource Identifier*, *URI*).
- Взаимодействие с ресурсами осуществляется с помощью стандартных методов протокола HTTP. Особенностью HTTP является возможность указать в запросе и ответе способ представления одного и того же ресурса по различным параметрам (например, по формату или кодировке).
- Ресурсы описывают себя сами. Вся информация, необходимая для обработки запроса к ресурсу, содержится в самом запросе.
- Ресурсы могут содержать ссылки на другие ресурсы.

В табл. 9 перечислены некоторые методы протокола HTTP, используемые в REST. Совокупность методов `PUT`, `GET`, `POST` и `DELETE` позволяет реализовать стандартный набор CRUD[2]-операций с ресурсом.

Таблица 9

Методы HTTP, используемые в REST

Метод	Действие	Безопасен?	Идемпотентен?
`GET`	Запрашивает указанное представление ресурса	Да	Да
`PUT`	Создаёт или обновляет ресурс	Нет	Да
`DELETE`	Удаляет ресурс	Нет	Да
`POST`	Посылает указанному ресурсу данные для обработки	Нет	Нет
`HEAD`	Аналог `GET`, но в ответе сервера отсутствует тело ресурса	Да	Да
`OPTIONS`	Возвращает список методов, поддерживаемых ресурсом	Да	Да

RESTful службы – это антипод традиционных веб-служб, основанных на вызове удаленных процедур (*Remote Procedure Call*, *RPC*). RPC позволяет использовать небольшое количество сетевых ресурсов с большим количеством методов и сложным протоколом. При подходе REST количество методов и сложность протокола строго ограничены, из-за чего количество отдельных ресурсов должно быть большим.

Использование REST рассмотрим на примере службы, управляющей веб-закладками (bookmarks). Служба хранит закладки зарегистрированного пользо-

[1] Термин *REST* (*Representational State Transfer*, *репрезентативная передача состояния*) ввёл Рой Томас Филдинг в своей диссертации в 2000 году.
[2] CRUD – Create, Read, Update, Delete.

вателя, позволяет разделить закладки на public- и private-категории, а также даёт возможность снабдить закладку произвольным набором тегов. Если применять традиционный RPC-подход, можно предположить наличие у веб-службы методов, перечисленных в табл. 10.

Таблица 10

Методы гипотетической RPC-службы управления закладками

Метод	Описание	Требует аутентификации?
`CreateUserAccount()`	Создает учетную запись пользователя	
`GetUserAccount()`	Получает данные учетной записи	Да
`UpdateUserAccount()`	Обновляет данные учетной записи	Да
`DeleteUserAccount()`	Удаляет учетную запись пользователя	Да
`GetUserProfile()`	Получает общедоступные данные учетной записи пользователя	
`CreateBookmark()`	Создает закладку	Да
`GetBookmark()`	Получает информацию о закладке	Да (для private-закладок)
`UpdateBookmark()`	Обновляет закладку	Да
`DeleteBookmark()`	Удаляет закладку	Да
`GetUserBookmarks()`	Получает все закладки пользователя (возможна фильтрация по тегу)	Да
`GetUserPublicBookmarks()`	Получает все public-закладки пользователя (возможна фильтрация по тегу)	
`GetPublicBookmarks()`	Получает все public-закладки всех пользователей (возможна фильтрация по тегу)	

Первый шаг проектирования RESTful службы – выявление ресурсов, которыми служба манипулирует. Анализ табл. 10 даёт следующий набор ресурсов.

1. Учетная запись пользователя.
2. Профиль пользователя.
3. Закладка.
4. Набор private-закладок конкретного пользователя.
5. Набор public-закладок конкретного пользователя.
6. Набор всех public-закладок.

Следующий шаг – создание схемы именования ресурсов. URI ресурса в веб – это URL, который обычно включает некую базовую часть (например, `http://bsuir.by/bookmarks`). Используем URL, состоящий только из базовой части, для списка всех public-закладок. Для остальных URI применим *шаблоны URL*, то есть небазовую часть URL будем рассматривать как шаблон с параметрами. Совокупность параметров однозначно идентифицирует ресурс. Например, шаблон `/users/{username}/bookmarks` содержит параметр `{username}` и указывает на private-закладки конкретного пользователя. Табл. 11 описывает шаблоны, применяемые в RESTful службе закладок.

Таблица 11

Шаблоны URL для идентификации ресурсов RESTful службы

Имя ресурса	Шаблон
Учетная запись пользователя	`/users/{username}`
Профиль пользователя	`/users/{username}/profile`
Закладка	`/users/{username}/bookmarks/{id}`
Набор private-закладок пользователя	`/users/{username}/bookmarks?tag={tag}`
Набор public-закладок пользователя	`/{username}?tag={tag}`
Набор всех public-закладок	`/?tag={tag}`

После разработки схемы именования, необходимо определить, как и какие методы HTTP будут выполнять манипуляции с ресурсами. Получение информации логично возложить на HTTP-метод `GET`. Если URI содержит некорректные параметры (например, имя несуществующего пользователя), будем возвращать HTTP-ответ с кодом 404 (`"Not found"`). Для создания пользователя применим метод `PUT`. Если пользователь создан успешно, будем возвращать код 201 (`"Created"`), а если пользователь с таким именем уже существует – код 401 (`"Unauthorized"`). Создание закладки обеспечит метод `POST`, применённый к ресурсу `/users/{username}/bookmarks`. В этом случае не применяется метод `PUT`, так как идентификатор закладки нам не известен. Обновление и удаление ресурсов обеспечат методы `PUT` и `DELETE` соответственно.

Таблица 12

Интерфейс REST-службы управления закладками

Метод RPC-службы	Шаблон URI	Метод HTTP
`CreateUserAccount()`	`/users/{username}`	`PUT`
`GetUserAccount()`	`/users/{username}`	`GET`
`UpdateUserAccount()`	`/users/{username}`	`PUT`
`DeleteUserAccount()`	`/users/{username}`	`DELETE`
`GetUserProfile()`	`/users/{username}/profile`	`GET`
`CreateBookmark()`	`/users/{username}/bookmarks`	`POST`
`GetBookmark()`	`/users/{username}/bookmarks/{id}`	`GET`
`UpdateBookmark()`	`/users/{username}/bookmarks/{id}`	`PUT`
`DeleteBookmark()`	`/users/{username}/bookmarks/{id}`	`DELETE`
`GetUserBookmarks()`	`/users/{username}/bookmarks?tag={tag}`	`GET`
`GetUserPublicBookmarks()`	`{username}?tag={tag}`	`GET`
`GetPublicBookmarks()`	`?tag={tag}`	`GET`

Некоторые методы нашей службы требуют аутентификации. Существуют различные подходы обеспечения безопасности при использовании REST. Наиболее распространён метод, при котором в запросы внедряется заголовок `Authorization` с информацией об имени пользователя и хешем секретного ключа. Этот секретный ключ генерируется и сохраняется на сервере при регистрации пользователя, а затем сообщается пользователю. И клиент, и сервер используют хеш ключа, чтобы не пересылать его в открытом виде.

Последним шагом проектирования нашей службы является выбор форматов представления ресурсов. В нашем случае будем считать базовым форматом XML. Можно предусмотреть использование других форматов, если расширить

схему шаблонов URL. Например, добавляя к шаблону параметр `?format=json`, мы сообщаем службе, что ожидаем данных в формате JSON.

Перейдём к вопросам практической реализации RESTful службы. Отметим, что для этих целей можно приспособить инфраструктуру ASP.NET, написав класс-обработчик, реализующий интерфейс `IHttpHandler`. В таком обработчике требуется проанализировать URL запроса (при помощи регулярных выражений или иного механизма) и метод запроса, а затем сформировать ответ сервера. Однако WCF даёт более удобный способ работы с REST, доступный начиная с версии WCF 3.5.

Начнём с описания контрактов данных для наших ресурсов.

```
public class User
{
    public Uri Id { get; set; }
    public string Username { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
    public Uri Bookmarks { get; set; }
}

public class UserProfile
{
    public Uri Id { get; set; }
    public string Name { get; set; }
    public Uri Bookmarks { get; set; }
}

public class Bookmark
{
    public Uri Id { get; set; }
    public string Title { get; set; }
    public Uri Url { get; set; }
    public string User { get; set; }
    public Uri UserLink { get; set; }
    public string Tags { get; set; }
    public bool Public { get; set; }
    public DateTime LastModified { get; set; }
}

[CollectionDataContract]
public class Bookmarks : List<Bookmark>
{
    public Bookmarks() { }

    public Bookmarks(List<Bookmark> bookmarks) : base(bookmarks) { }
}
```

Начиная с .NET Framework 3.5 SP1 при описании контракта данных не требуется явно указывать атрибуты [DataContract] и [DataMember]. При этом элементы класса рассматриваются как необязательные: десериализация проходит, даже если некоторые из элементов отсутствуют в потоке данных.

Далее следует создать сервисный контракт. Для простоты изложения используем для этого класс, а не интерфейс. При описании сервисного контракта RESTful службы к методам можно применить атрибуты [WebGet] и [WebInvoke] (пространство имён System.ServiceModel.Web, одноимённая сборка). Атрибут [WebInvoke] имеет строковое свойство Method, указывающее на связанный HTTP-метод. У обоих атрибутов есть свойство UriTemplate для задания строки с шаблоном URL. Если имена параметров в шаблоне совпадают с именами параметров метода, WCF выполнит автоматическое связывание соответствующих значений. Для методов контракта, обрабатывающих PUT и POST запросы, необходимо последним параметром указать объект, в который десериализуется тело запроса. Ниже показана «заготовка» сервисного контракта с пустыми методами.

```
[ServiceContract]
public class BookmarkService
{
    [WebGet(UriTemplate = "?tag={tag}")]
    [OperationContract]
    public Bookmarks GetPublicBookmarks(string tag) { }

    [WebGet(UriTemplate = "/{username}?tag={tag}")]
    [OperationContract]
    public Bookmarks GetUserPublicBookmarks(string username,
                                            string tag) { }

    [WebGet(UriTemplate = "/users/{username}/bookmarks?tag={tag}")]
    [OperationContract]
    public Bookmarks GetUserBookmarks(string username,
                                      string tag) { }

    [WebGet(UriTemplate = "/users/{username}/profile")]
    [OperationContract]
    public UserProfile GetUserProfile(string username) { }

    [WebGet(UriTemplate = "/users/{username}")]
    [OperationContract]
    public User GetUser(string username) { }

    [WebGet(UriTemplate = "/users/{username}/bookmarks/{id}")]
    [OperationContract]
    public Bookmark GetBookmark(string username, string id) { }

    [WebInvoke(Method = "PUT", UriTemplate = "/users/{username}")]
    [OperationContract]
    public void PutUserAccount(string username, User user) { }
```

```
    [WebInvoke(Method = "DELETE", UriTemplate ="/users/{username}")]
    [OperationContract]
    public void DeleteUserAccount(string username) { }

    [WebInvoke(Method = "POST",
               UriTemplate = "/users/{username}/bookmarks")]
    [OperationContract]
    public void PostBookmark(string username, Bookmark newValue){ }

    [WebInvoke(Method = "PUT",
               UriTemplate = "/users/{username}/bookmarks/{id}")]
    [OperationContract]
    public void PutBookmark(string username, string id,
                            Bookmark bm) { }

    [WebInvoke(Method = "DELETE",
               UriTemplate = "/users/{username}/bookmarks/{id}")]
    [OperationContract]
    public void DeleteBookmark(string username, string id) { }
}
```

При реализации методов RESTful службы оказывается полезным класс `WebOperationContext`. С помощью этого класса можно получить доступ к данным и заголовкам HTTP-сообщений. Следующий пример демонстрирует использование `WebOperationContext` в методе `GetUser()`.

```
[WebGet(UriTemplate = "/users/{username}")]
[OperationContract]
public User GetUser(string username)
{
    // статическое свойство Current указывает на текущий контекст
    var response = WebOperationContext.Current.OutgoingResponse;

    // ниже представлен «набросок» функции IsUserAuthorized()
    if (!IsUserAuthorized(username))
    {
        response.StatusCode = HttpStatusCode.Unauthorized;
        return null;
    }

    // функция FindUser() получает информацию из хранилица данных
    User user = FindUser(username);
    if (user == null)
    {
        response.SetStatusAsNotFound();
    }
    return user;
}
```

```
private bool IsUserAuthorized(string username)
{
    var request = WebOperationContext.Current.IncomingRequest;
    string auth = request.Headers[HttpRequestHeader.Authorization];
    // далее должен следовать код, который выделяет имя пользователя
    // и хеш пароля и выполянет необходимые проверки
    . . .
}
```

Атрибуты [WebGet] и [WebInvoke] имеют свойства RequestFormat и ResponseFormat для настройки форматов сообщений HTTP. Тип свойств - перечисление WebMessageFormat, которое имеет два элемента: Xml и Json.

Для хостинга RESTful службы WCF следует настроить концевую точку. Как и для обычных компонентов WCF, это можно сделать программно или при помощи файла конфигурации. Особенностью является использование новой привязки WebHttpBinding совместно с поведением WebHttpBehavior. Ниже представлен фрагмент конфигурационного файла и код, запускающий хостинг.

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="BookmarkService">
       <endpoint binding="webHttpBinding" contract="BookmarkService"
                 behaviorConfiguration="webHttp"/>
      </service>
    </services>
    <behaviors>
      <endpointBehaviors>
        <behavior name="webHttp">
          <webHttp/>
        </behavior>
      </endpointBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>

var host = new ServiceHost(typeof (BookmarkService),
                  new Uri("http://localhost:8080/bookmarkservice"));
host.Open();
```

Для упрощения организации хостинга RESTful службы предназначен класс WebServiceHost – наследник класса ServiceHost. Используя WebServiceHost, достаточно указать тип службы и адрес концевой точки. Необходимость в какой-либо дополнительной конфигурации отпадает!

```
// конфигурационный файл не используется!
var host = new WebServiceHost(typeof (BookmarkService),
                  new Uri("http://localhost:8080/bookmarkservice"));
host.Open();
```

Если для хостинга RESTful службы применяется ASP.NET, используется класс `WebServiceHostFactory`, указываемый как значение атрибута `Factory` в директиве `@ServiceHost`. Конфигурацию службы в `web.config` нужно или удалить, или изменить привязку на `webHttpBinding`.

```
<%@ ServiceHost Language="C#" Debug="true" Service="BookmarkService"
   Factory="System.ServiceModel.Activation.WebServiceHostFactory" %>
```

При хостинге в ASP.NET RESTful службу можно дополнительно настроить, включив режим совместимости с ASP.NET. Это откроет доступ из методов службы к объекту `HttpContext`. Для включения режима совместимости применяется параметр в конфигурационном файле и дополнительный атрибут для класса службы.

```
<configuration>
  <system.serviceModel>
    <serviceHostingEnvironment aspNetCompatibilityEnabled="true"/>
  </system.serviceModel>
</configuration>

[AspNetCompatibilityRequirements(RequirementsMode =
                        AspNetCompatibilityRequirementsMode.Allowed)]
[ServiceContract]
public class BookmarkService { }
```

Клиентские приложения RESTful службы могут быть созданы с применением различных технологий. Если клиентом является AJAX-приложение, следует использовать JavaScript и объект `XMLHttpRequest`. Если клиентское приложение разрабатывается на платформе .NET, то оно может задействовать классы `HttpWebResponse` и `HttpWebRequest` («сырая» работа с протоколом HTTP). Для применения на клиенте технологии WCF следует выделить сервисный контракт в отдельный интерфейс, а канал организовать на основе `WebChannelFactory`.

```
var cf = new WebChannelFactory<IBookmarkService>(
             new Uri("http://localhost:55555/BookmarkService.svc"));
IBookmarkService channel = cf.CreateChannel();
Bookmarks bms = channel.GetPublicBookmarks("WCF");
foreach (Bookmark bm in bms)
    Console.WriteLine("{0}\r\n{1}", bm.Title, bm.Url);
```