ЧАСТЬ VI
Коммуникации

В ЭТОЙ ЧАСТИ...

43

Технология Windows Communication Foundation

В ЭТОЙ ГЛАВЕ...

- Обзор WCF
- Простая служба и клиент
- Контракты служб, операций, данных и сообщений
- Реализация службы
- Использование привязки для обеспечения связи
- Разные варианты хостов для служб
- Создание клиентов за счет добавления ссылки на службы и программным образом
- Дуплексная связь

До выхода .NET 3.0 в одном и том же решении масштаба предприятия необходимо было использовать несколько коммуникационных технологий. Для обеспечения независимых от платформы коммуникаций применялись веб-службы ASP.NET. Для предоставления более совершенных веб-служб, обеспечивающих высокую надежность, безопасность и атомарные транзакции, в веб-службы ASP.NET был добавлен набор расширений Web Services Enhancements, который привнес дополнительный уровень сложности. Если требовалось обеспечить более быструю связь и применять приложения .NET на стороне клиента и службы, предпочтение отдавалось технологии .NET Remoting или .NET Enterprise Services, которые поддерживали возможность выполнения атомарных транзакций, по умолчанию использовали протокол DCOM и работали быстрее, чем .NET Remoting. DCOM был также единственным протоколом, который позволял осуществлять передачу транзакций. Все эти технологии поддерживали разные модели программирования, что требовало наличия у разработчиков множества навыков.

В .NET Framework 3.0 появилась новая коммуникационная технология под названием Windows Communication Foundation (WCF). Она включает в себя функциональные возможности всех своих предшественниц и объединяет их в единую модель программирования.

Обзор WCF

Технология WCF сочетает в себе функциональные возможности веб-служб ASP.NET, технологии .NET Remoting, Message Queuing и Enterprise Services. Применяя WCF, разработчик получает в свое распоряжение следующие средства.

- **Хосты для компонентов и служб**. Точно так же, как в .NET Remoting и Web Service Enhancements (WSE) можно применять специальные хосты, в WCF допускается использовать для службы в качестве хоста исполняющую среду ASP.NET, службу Windows, процесс COM+ или даже просто приложение Windows Forms для обеспечения одноранговых (peer-to-peer) вычислений.
- **Декларативное поведение**. Вместо требования обязательно наследовать службы от базового класса (которое существует в .NET Remoting и Enterprise Services), в WCF для определения служб можно использовать атрибуты, подобно тому, как это делается при разработке веб-служб ASP.NET.
- **Каналы связи**. Хотя технология .NET Remoting является очень гибкой в том, что касается изменения канала связи, WCF представляет собой замечательную альтернативу, поскольку в ней обеспечивается точно такая же степень гибкости. В WCF предлагается множество каналов для обеспечения коммуникаций через HTTP, TCP и IPC. Можно также создавать собственные каналы, работающие с другими транспортными протоколами.
- **Инфраструктура безопасности**. Для реализации независимых от платформы веб-служб должна обязательно использоваться стандартизированная среда безопасности. Предлагаемые стандарты были реализованы с помощью WSE 3.0, и это же касается WCF.
- **Расширяемость**. В .NET Remoting предлагается много возможностей для расширения. Можно не только создавать специальные каналы, форматировщики и прокси, но также и внедрять функциональность внутрь потока сообщений как на стороне клиента, так и на стороне сервера. В WCF предлагаются похожие возможности, но здесь расширения должны создаваться с применением заголовков SOAP.
- **Поддержка предыдущих технологий**. Вместо того чтобы полностью переписывать распределенное решение для использования WCF, можно интегрировать WCF в существующие технологии. В WCF предлагается канал, который может обеспечивать связь с обслуживаемыми компонентами через DCOM. Веб-службы, которые разрабатывались с помощью ASP.NET, тоже могут интегрироваться с WCF.

Конечной целью является налаживание отправки и приема сообщений между клиентом и службой среди различных процессов или систем, по локальной сети или через Интернет. Происходить все это должно, если требуется, не зависящим от платформы образом и насколько возможно быстро. С удаленной точки зрения служба предоставляет конечную точку, которая описана с помощью контракта, привязки и адреса. В контракте перечисляются предлагаемые службой операции, в привязке предоставляется информация о протоколе и кодировании, а в адресе указывается местонахождения службы. Для получения доступа к службе клиент должен иметь совместимую конечную точку.

На рис. 43.1 показаны компоненты, принимающие участие в коммуникациях WCF.

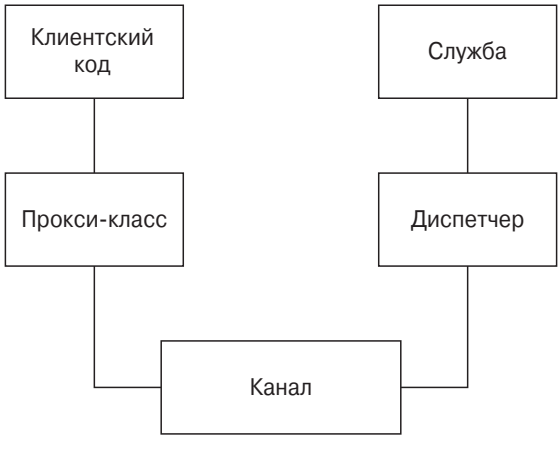

Рис. 43.1. *Компоненты, принимающие участие в коммуникациях WCF*

Клиент вызывает какой-то метод прокси-класса. Прокси-класс предлагает методы, предоставляемые службой, но их вызов преобразуется в сообщение, которое затем передается по каналу. Канал имеет клиентскую и серверную части, взаимодействующие между собой через сетевой протокол. Из канала сообщение передается диспетчеру, который преобразует его обратно в вызов метода, после чего указанный метод вызывается в службе.

В WCF поддерживается несколько коммуникационных протоколов. Для обеспечения независимых от платформы коммуникаций поддерживаются стандартные протоколы веб-служб, а для обеспечения связи между приложениями .NET — более быстрые протоколы с меньшим количеством накладных расходов.

В следующих разделах рассматривается ключевая функциональность, которая включает перечисленные ниже средства.

- SOAP — независимый от платформы протокол, который является основой множества спецификаций веб-служб и обеспечивает безопасность, транзакции и надежность.
- WSDL (Web Services Description Language — язык описания веб-служб) — язык, предоставляющий метаданные для описания служб.
- REST (Representational State Transfer — передача репрезентативного состояния) — протокол, используемый вместе с веб-службами REST для коммуникаций через HTTP.
- JSON (JavaScript Object Notation — представление объектов JavaScript) — нотация, упрощающая работу с клиентами JavaScript.

Протокол SOAP

Для обеспечения независимых от платформы коммуникаций можно использовать протокол SOAP, который поддерживается в WCF напрямую. Первоначально название SOAP представляло собой сокращение от Simple Object Access Protocol (Простой протокол доступа к объектам), но, начиная с версии SOAP 1.2, перестало быть таковым. Протокол SOAP больше не является протоколом для доступа к объектам, потому что вместо этого позволяет пересылать сообщения, определяемые с помощью схемы XML.

Служба принимает сообщение SOAP от клиента и в ответ тоже отправляет сообщение SOAP. Каждое SOAP-сообщение включает в себя "конверт", в котором содержится заголовок и тело:

```
<s:Envelope xmlns:a="http://www.w3.org/2005/08/addressing"
    xmlns:s="http://www.w3.org/2003/05/soap-envelope">
  <s:Header>
  </s:Header>
  <s:Body>
    <ReserveRoom xmlns="http://www.wrox.com/ProCSharp/2010">
     <roomReservation
        xmlns:d4p1="http://schemas.datacontract.org/2009/10/Wrox.ProCSharp.WCF"
        xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
       <d4p1:RoomName>Hawelka</d4p1:RoomName>
       <d4p1:StartDate>2010-06-21T08:00:00</d4p1:StartDate>
       <d4p1:EndDate>2010-06-21T14:00:00</d4p1:EndDate>
       <d4p1:Contact>Georg Danzer</d4p1:Contact>
       <d4p1:Event>White Horses</d4p1:Event>
     </roomReservation>
    </ReserveRoom>
  </s:Body>
</s:Envelope>
```

Заголовок является необязательным и может содержать информацию об адресах, безопасности и транзакциях. В теле содержатся собственно данные сообщения.

Документ WSDL

В документе WSDL описаны операции и сообщения службы. Язык WSDL определяет метаданные службы, которые могут использоваться для создания необходимого клиентскому приложению прокси-класса.

В документе WSDL содержится следующая информация.

- **Типы** сообщений, которые описываются с помощью схемы XML.
- **Сообщения**, которые могут пересылаться службе и из нее. Частью этих сообщений являются типы, описанные с помощью схемы XML.
- **Типы портов**, которые отображаются на контракты служб и отражают список операций, определяемых с помощью контракта службы. Операции содержат сообщения; например, входные и выходные сообщения, используемые в последовательности запроса и ответа.
- **Информация о привязке**, в которой содержатся операции, перечисленные вместе с типами портов, и указано, как должен использоваться вариант SOAP.
- **Информация о службе**, в которой типы портов отображаются на адреса конечных точек.

В WCF информация WSDL предоставляется конечными точками MEX (Metadata Exchange — обмен метаданными).

Протокол REST

В WCF можно организовать коммуникации с применением протокола REST. В действительности REST протоколом не является, но определяет несколько принципов, которыми можно руководствоваться при использовании служб для получения доступа к ресурсам. Веб-службой REST называется такая веб-служба, которая основана на применении протокола HTTP и принципов REST. Эти принципы предполагают три вещи: возможность получения к службе доступа с помощью простого URI, поддержку в службе типов MIME и применение различных методов HTTP. Благодаря поддержке типов MIME, данные из службы могут возвращаться в разных форматах, например, простой XML, JSON или AtomPub. За возврат данных из службы в HTTP-запросе отвечает метод `GET`. К числу других используемых методов относятся `PUT`, `POST` и `DELETE`. Метод `PUT` применяется для внесения обновления на стороне службы, метод `POST` — для создания нового ресурса, а метод `DELETE` — для удаления ресурса.

REST позволяет отправлять службам не такие большие запросы, как SOAP. Если необходимость в транзакциях, защите сообщений (но не защите самих коммуникаций, например, посредством HTTPS) и надежности, которые обеспечивает SOAP, отсутствует, служба с архитектурой REST может значительно сократить накладные расходы.

В случае применения архитектуры REST служба всегда не имеет состояния, и ответ от нее может помещаться в кэш.

Нотация JSON

Вместо отправки сообщений SOAP доступ к службам из кода JavaScript лучше всего осуществлять с использованием JSON. В .NET имеется класс сериализации контрактов данных, который позволяет создавать объекты с помощью нотации JSON.

В случае применения JSON накладные расходы получаются меньше, чем для SOAP, поскольку JSON не является нотацией XML, а оптимизирована под клиентов JavaScript. Это делает ее исключительно удобной для использования в клиентах Ajax. Технология Ajax подробно рассматривается в главе 41. Возможностей для обеспечения надежности, безопасности и транзакций, которые могут пересылаться в заголовке SOAP, нотация JSON не предоставляет, но в клиентах JavaScript подобные средства обычно не нужны.

Простая служба и клиент

Прежде чем переходить к рассмотрению деталей WCF, давайте создадим простую службу, которая будет предназначена для резервирования комнат под совещания.

В качестве фонового хранилища для данных о резервировании комнат применяется простая база данных SQL Server с таблицей `RoomReservations`. Эта база данных входит в состав примеров кода для этой главы.

Создайте пустое решение по имени `RoomReservation` и добавьте в него новый проект Component Library, назначив ему имя `RoomReservationData`. В этом первом реализуемом проекте содержится только код для доступа к базе данных. Поскольку ADO.NET Entity Framework существенно упрощает код для доступа к базе данных, именно эта технология .NET 4 здесь и применяется.

Более подробно ADO.NET Entity Framework рассматривается в главе 31.

Добавьте новый элемент ADO.NET Entity Data Model (Модель сущностных данных ADO.NET) и назначьте ему имя `RoomReservationModel.edmx`. Чтобы создать сущностную модель данных, выберите базу данных `RoomReservations` и затем таблицу

RoomReservations. После ее создания в окне визуального конструктора появится сущность RoomReservation, как показано на рис. 43.2. Визуальный конструктор создаст класс сущности RoomReservation, содержащий свойства для каждого из столбцов соответствующей таблицы, а также класс RoomReservationDataContext, содержащий код для установки соединения с базой данных и регистрации классов сущностей для получения уведомления обо всех изменениях.

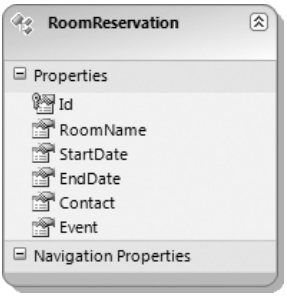

Рис. 43.2. Класс сущности RoomReservation

Для обеспечения чтения и записи данных в базу с помощью ADO.NET Entity Framework понадобится добавить класс RoomReservationData. Метод ReserveRoom() в этом классе будет вносить запись о резервировании комнаты в базу данных, а метод GetReservations() — возвращать массив записей о резервировании комнат за указанный диапазон дат.

```
using System;
using System.Collections.Generic;
using System.Linq;
namespace Wrox.ProCSharp.WCF
{
  public class RoomReservationData
  {
    public void ReserveRoom(RoomReservation roomReservation)
    {
      using (var data = new RoomReservationsEntities())
      {
        data.RoomReservations.AddObject(roomReservation);
        data.SaveChanges();
      }
    }
    public RoomReservation[] GetReservations(DateTime fromDate, DateTime toDate)
    {
      using (var data = new RoomReservationsEntities())
      {
        return (from r in data.RoomReservations
         where r.StartDate > fromDate && r.EndDate < toDate select r).ToArray();
      }
    }
  }
}
```

*Фрагмент кода **RoomReservationData\RoomReservationData.cs***

Теперь можно переходить к созданию самой службы.

Контракт службы

Добавьте в решение новый проект типа WCF Service Library и назначьте ему имя RoomReservationService.

Переименуйте сгенерированные в нем файлы IService1.cs в IRoomService.cs и Service1.cs в RoomReservationService.cs и замените пространство имен внутри них на Wrox.ProCSharp.WCF. Добавьте в него ссылку на сборку RoomReservationData, чтобы сделать доступными сущностные типы и класс RoomReservationData.

Предлагаемые службой операции могут определяться в интерфейсе. Здесь внутри интерфейса IRoomService определены методы ReserveRoom и GetRoomReservations. Контракт службы определяется с помощью атрибута ServiceContract, а определяемые в нем операции службы снабжаются атрибутом OperationContract.

```
using System;
using System.ServiceModel;
namespace Wrox.ProCSharp.WCF
{
  [ServiceContract()]
  public interface IRoomService
  {
    [OperationContract]
    bool ReserveRoom(RoomReservation roomReservation);

    [OperationContract]
    RoomReservation[] GetRoomReservations(DateTime fromDate, DateTime toDate);
  }
}
```

*Фрагмент кода **RoomReservationService\IRoomService.cs***

Реализация службы

Класс службы `RoomReservationService` реализует интерфейс `IRoomService`. Сама служба реализуется просто вызовом надлежащих методов класса `RoomReservationData`.

```
using System;
using System.Linq;

namespace Wrox.ProCSharp.WCF
{
  public class RoomReservationService: IRoomService
  {
    public bool ReserveRoom(RoomReservation roomReservation)
    {
      var data = new RoomReservationData();
      data.ReserveRoom(roomReservation);
      return true;
    }
    public RoomReservation[] GetRoomReservations(DateTime fromDate,
                                                 DateTime toDate)
    {
      var data = new RoomReservationData();
      return data.GetReservations(fromDate, toDate);
    }
  }
}
```

*Фрагмент кода **RoomReservationService\RoomReservationService.cs***

Хост и тестовый клиент службы WCF

При использовании шаблона проекта WCF Service Library создается конфигурационный файл приложения по имени `App.config`, который необходимо адаптировать под имена нового класса и интерфейса. Элемент `service` ссылается на тип службы `RoomReservationService`, включая пространство имен, а интерфейс контракта нуждается в определении элемента `endpoint`.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
 <system.web>
  <compilation debug="true" />
 </system.web>
```

```
<!-- При развертывании проекта библиотеки службы содержимое конфигурационного файла
     должно быть добавлено в файл хоста app.config, поскольку System.Configuration
     не поддерживает использование конфигурационных файлов для библиотек. -->
<system.serviceModel>
 <services>
 <service name="Wrox.ProCSharp.WCF.RoomReservationService">
  <host>
   <baseAddresses>
    <add baseAddress =
     "http://localhost:8732/Design_Time_Addresses/RoomReservationService/Service1/"
    />
   </baseAddresses>
  </host>
 <!-- Конечные точки службы -->
 <!-- Если адрес не указан полностью, он берется относительно
      базового адреса, который был предоставлен выше -->
 <endpoint address ="" binding="wsHttpBinding"
           contract="Wrox.ProCSharp.WCF.IRoomService">
 <!-- При развертывании идущий далее элемент идентификационных данных должен
      быть удален или заменен так, чтобы он отражал идентификационные данные, с
      использованием которых должна выполняться развернутая служба. Если его удалить,
      WCF будет вставлять необходимые идентификационные данные автоматически. -->
  <identity>
    <dns value="localhost" />
   </identity>
  </endpoint>
  <!-- Конечные точки метаданных -->
  <!-- Конечная точка Metadata Exchange используется службой для описания себя
       самой клиентам. -->
  <!-- Эта конечная точка не предусматривает использование безопасной привязки
       и потому перед развертыванием должна быть защищена или удалена -->
  <endpoint address="mex" binding="mexHttpBinding" contract="IMetadataExchange" />
 </service>
</services>
<behaviors>
 <serviceBehaviors>
  <behavior>
    <!-- Во избежание раскрытия информации о метаданных, перед развертыванием
         идущее ниже значение должно быть установлено в false, а расположенная
         выше конечная точка метаданных — удалена -->
    <serviceMetadata httpGetEnabled="True" />
    <!-- Для получения детальной информации в ошибках в целях отладки идущее ниже
         значение должно быть установлено в true, но перед развертыванием оно должно
         быть снова установлено в false во избежание раскрытия деталей исключений -->
    <serviceDebug includeExceptionDetailInFaults="False" />
   </behavior>
  </serviceBehaviors>
 </behaviors>
</system.serviceModel>
</configuration>
```

Фрагмент кода ***RoomReservationService\App.config***

С адресом службы `http://localhost:8731/Design_Time_Addresses` ассоциирован список контроля доступа (access control list – ACL), который позволяет интерактивному пользователю создавать порт-слушатель. По умолчанию пользователю, не являющемуся администратором, не разрешено открывать порты в режиме прослушивания. Просматривать списки ACL можно с помощью команды `netsh http show urlacl`, а добавлять в них новые записи – с помощью команды `netsh http add urlacl url=http://+:8080/MyURI user=someUser`.

При запуске этой библиотеки из Visual Studio 2010 запустится хост службы WCF (WCF Service Host), о чем будет свидетельствовать появление соответствующего значка в области уведомлений панели задач. Щелчок на этом значке приводит к открытию диалогового окна WCF Service Host (рис. 43.3), в котором можно просмотреть состояние службы. В окне свойств проекта приложения WCF Library доступна специальная вкладка, посвященная WCF, с опциями, одна из которых позволяет указать, должен ли хост службы WCF запускаться при запуске проекта из того же решения. По умолчанию эта опция включена. Кроме того, в окне свойств проекта в разделе конфигурации Debug (Отладка) можно увидеть определенный аргумент `/client:"WcfTestClient.exe"`. Наличие этого аргумента позволяет хосту службы WCF запускать тестовое клиентское приложение (WCF Test Client), как показано на рис. 43.4, которое позволяет проверить функционирование службы. Двойной щелчок на какой-то операции в окне этого тестового клиента вызывает отображение в его правой его части полей для ввода данных, в которых можно указать данные для отправки службе. Перейдя на вкладку XML в этом окне, можно просмотреть все отправленные и полученные службой сообщения SOAP.

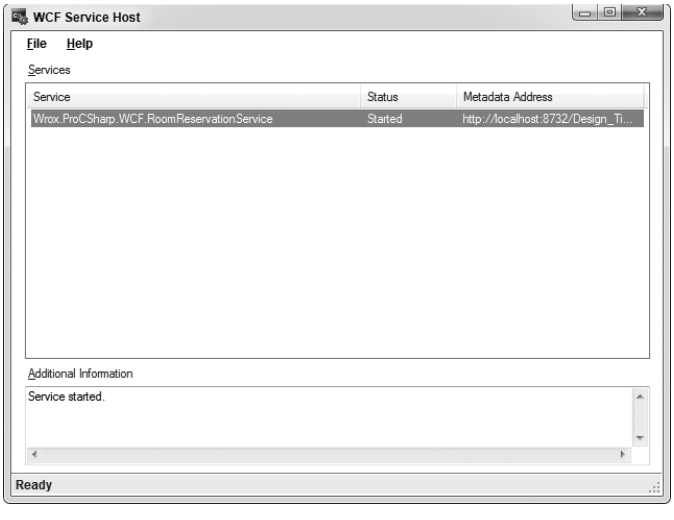

Рис. 43.3. *Окно WCF Service Host*

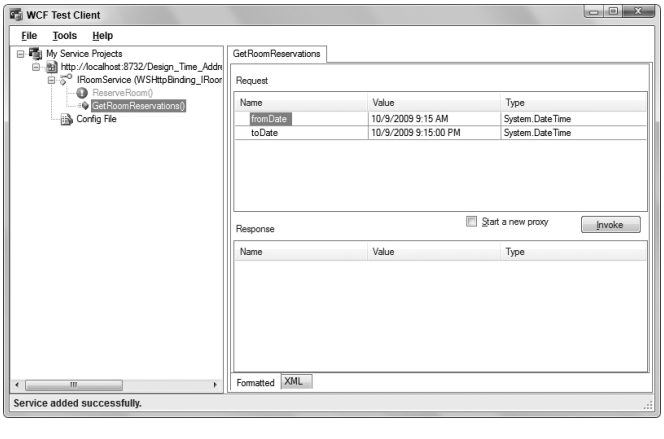

Рис. 43.4. *Окно WCF Test Client*

Специальный хост службы

В WCF имеется возможность запускать службы в любом хосте. Можно создать приложение Windows Forms или WPF для одноранговых служб (peer-to-peer services), создать службу Windows или применить в качестве хоста компонент Windows Activation Services (WAS). Для демонстрации простого хоста прекрасно подходит также и консольное приложение.

В хосте для нашей службы необходимо сослаться на библиотеку `RoomReservation Service`. Запуск службы производится за счет создания экземпляра и открытия объекта типа `ServiceHost`. Этот класс содержится в пространстве имен `System.ServiceModel`. В его конструкторе указывается класс `RoomReservationService`, который реализуют эту службу. Вызов метода `Open()` приводит к открытию канала-слушателя службы и тем самым позволяет службе начать принимать запросы. Метод `Close()` закрывает этот канал.

```
using System;
using System.ServiceModel;

namespace Wrox.ProCSharp.WCF
{
  class Program
  {
    internal static ServiceHost myServiceHost = null;
    internal static void StartService()
    {
      myServiceHost = new ServiceHost(typeof(RoomReservationService));
      myServiceHost.Open();
    }
    internal static void StopService()
    {
      if (myServiceHost.State != CommunicationState.Closed)
        myServiceHost.Close();
    }
    static void Main()
    {
      StartService();
      Console.WriteLine("Сервер запущен. Для завершения нажмите <Enter>");
      Console.ReadLine();
      StopService();
    }
  }
}
```

Фрагмент кода ***RoomReservationServiceHost\Program.cs***

Для конфигурирования WCF необходимо скопировать конфигурационный файл приложения, который был создан вместе с библиотекой службы для приложения-хоста. Затем его можно отредактировать с помощью программы WCF Service Configuration Editor (Редактор конфигурации служб WCF), окно которой показано на рис. 43.5. Если используется специальный хост для службы, можно снять отметку с опции WCF, указывающей запускать хост службы WCF, в параметрах проекта библиотеки WCF.

Клиент WCF

Для создания клиента в WCF можно использовать приложения многих разных типов и простое консольное приложение в том числе. Для рассматриваемого примера с резервированием комнат будет создано приложение WPF с элементами управления, показанное на рис. 43.6.

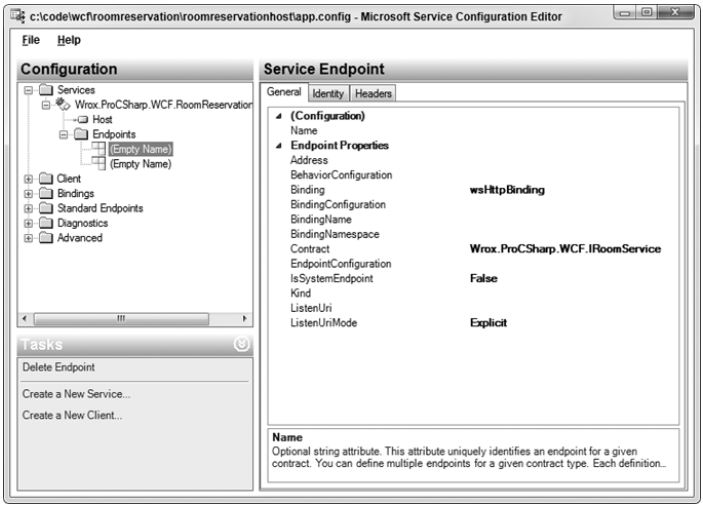

Рис. 43.5 *Окно редактора WCF Service Configuration Editor*

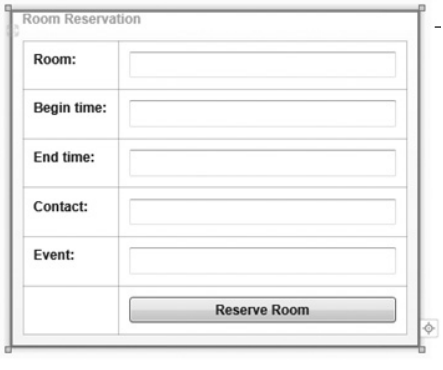

Рис. 43.6. *Клиентское приложение WPF для резервирования комнат*

Поскольку служба предоставляет конечную точку MEX с привязкой `mexHttpBinding` и доступ к метаданным включен в конфигурации, удобнее всего добавить ссылку на службу в Visual Studio. При добавлении ссылки на службу появляется диалоговое окно Add Service Reference (Добавление ссылки на службу), показанное на рис. 43.7. Щелчок на кнопке Discover (Обнаружить) позволяет найти все службы, доступные внутри данного решения.

Введите путь к службе и назначьте ссылке на службу имя `RoomReservationService`. Имя этой ссылки определяет пространство имен генерируемого прокси-класса.

Добавление ссылки на службу приводит к добавлению ссылок на сборки `System.Runtime.Serialization` и `System.ServiceModel` и конфигурационного файла, в котором содержатся данные относительно привязки и адрес конечной точки службы.

Класс `RoomReservation` генерируется из контракта данных. В этом классе содержатся все элементы `[DataMember]` из этого контракта. Класс `RoomServiceClient` генерируется как прокси-класс для клиента и содержит методы, определенные в контракте службы. С использованием этого клиента работающей службе можно отправлять заявки на резервирование комнат.

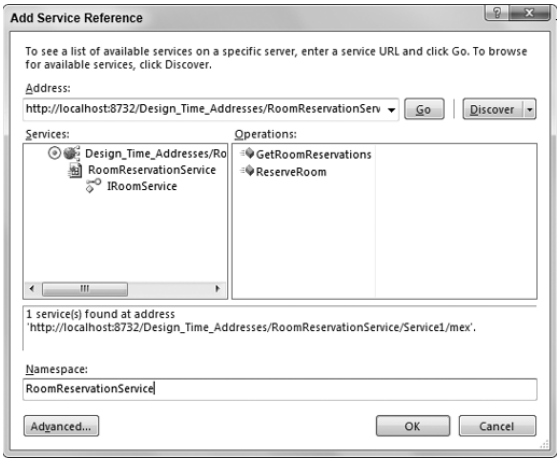

Рис. 43.7. *Диалоговое окно* Add Service Reference

```
private void OnReserveRoom(object sender, RoutedEventArgs e)
{
  RoomReservation reservation = new RoomReservation()
  {
    var = textRoom.Text,
    Event = textEvent.Text,
    Contact = textContact.Text,
    StartDate = DateTime.Parse(textStartTime.Text),
    EndDate = DateTime.Parse(textEndTime.Text)
  };
   var client = new RoomServiceClient();
   bool reserved = client.ReserveRoom(reservation);
   client.Close();
   if (reserved)
   MessageBox.Show("reservation ok");
                 // Резервирование прошло успешно
}
```

Фрагмент кода ***RoomReservationClient\MainWindow.xaml.cs***

Запустив одновременно службу и клиентское приложение, заявки на резервирование комнат можно добавлять в базу данных. В параметрах решения `RoomReservation` можно сделать стартовыми несколько проектов, каковыми в данном случае должны быть `RoomReservationClient` и `RoomReservationHost`.

Диагностика

При запуске приложения клиента и службы часто полезно знать о том, что происходит "за кулисами". Для этого в WCF предлагается источник трассировки, который понадобится просто соответствующим образом сконфигурировать. Настроить трассировку можно с помощью редактора Service Configuration Editor, щелкнув в нем на узле Diagnostics (Диагностика) и затем на элементах Enable Tracing (Включить трассировку) и Enable Message Logging (Включить протоколирование сообщений). Установка для уровня трассировки источников значения Verbose (Расширенный) позволяет получать максимально детальную информацию. Выполнение такой настройки приводит к добавлению в конфигурационный файл приложения источников трассировки и слушателей, как показано ниже.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
 <system.diagnostics>
  <sources>
   <source name="System.ServiceModel.MessageLogging"
           switchValue="Verbose,ActivityTracing">
    <listeners>
     <add type="System.Diagnostics.DefaultTraceListener" name="Default">
      <filter type="" />
     </add>
     <add name="ServiceModelMessageLoggingListener">
      <filter type="" />
     </add>
    </listeners>
   </source>
   <source name="System.ServiceModel" switchValue="Verbose,ActivityTracing"
    propagateActivity="true">
    <listeners>
     <add type="System.Diagnostics.DefaultTraceListener" name="Default">
      <filter type="" />
     </add>
    <add name="ServiceModelTraceListener">
    <filter type="" />
   </add>
    </listeners>
   </source>
  </sources>
  <sharedListeners>
    <add initializeData="c:\code\wcf\roomreservation\roomreservationhost\app_
messages.svclog"
     type="System.Diagnostics.XmlWriterTraceListener, System, Version=4.0.0.0,
           Culture=neutral, PublicKeyToken=b77a5c561934e089"
     name="ServiceModelMessageLoggingListener" traceOutputOptions="Timestamp">
     <filter type="" />
    </add>
    <add initializeData="c:\code\wcf\roomreservation\roomreservationhost\app_
tracelog.svclog"
     type="System.Diagnostics.XmlWriterTraceListener, System, Version=4.0.0.0,
      Culture=neutral, PublicKeyToken=b77a5c561934e089"
     name="ServiceModelTraceListener" traceOutputOptions="Timestamp">
     <filter type="" />
    </add>
  </sharedListeners>
  <trace autoflush="true" />
</system.diagnostics>
<!-- -->
```

Фрагмент кода ***RoomReservationHost\App.config***

В реализации классов WCF для записи трассировочных сообщений используются источники трассировки `System.ServiceModel` *и* `System.ServiceModel.MessageLogging`. *Более подробные сведения о трассировке и конфигурировании источников трассировки и слушателей можно найти в главе 19.*

Если установлен режим Verbose, то после запуска приложения файлы трассировки очень быстро разрастаются в размере. Для анализа информации из XML-файлов журналов в .NET SDK предлагается утилита Service Trace Viewer (Просмотр данных трассировки службы) — `svctraceviewer.exe`. На рис. 43.8 показано, как может выглядеть окно этой утилиты после включения трассировки и протоколирования сообщений. В конфигурации

по умолчанию отображается несколько видов сообщений; многие из них касаются безопасности. В зависимости от существующих требований в плане безопасности могут выбираться и другие параметры конфигурации.

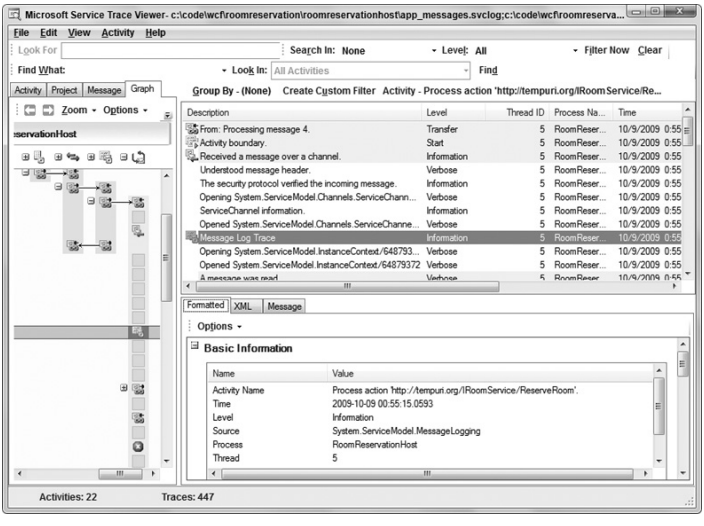

Рис. 43.8. *Окно утилиты Service Trace Viewer*

В следующих разделах рассматриваются детали и различные опции WCF.

Контракты

Контракт определяет функциональность, предлагаемую службой, и функциональные возможности, которые могут использоваться клиентом. Контракт может совершенно не зависеть от реализации службы.

Контракты, определяемые в WCF, делятся на три вида: контракты данных, контракты служб и контракты сообщений. Все они указываются с применением атрибутов .NET.

- **Контракт данных**. В контракте данных определяются данные, которые может принимать и возвращать служба. Классы, используемые для отправки и получения сообщений, имеют ассоциированные с ними атрибуты контрактов данных.
- **Контракт службы**. Контракт службы применяется для определения WSDL-документа, описывающего службу. Этот контракт определяется вместе с интерфейсами или классами.
- **Контракт сообщений**. Если необходим полный контроль над поведением SOAP-сообщений, в контракте сообщений можно указать, какие данные должны идти в заголовке, а какие — в теле SOAP-сообщений.

В следующих разделах более подробно рассказывается обо всех перечисленных видах контрактов, а также о деталях управления версиями, которые должны быть обязательно продуманы при их определении.

Контракт данных

В контракте данных типы CLR отображаются на схемы XML. Контракт данных отличается от прочих механизмов сериализации в .NET: в случае сериализации исполняю-

щей среды сериализуются все в поля (в том числе и приватные), а в случае сериализации XML — только общедоступные поля и свойства. Контракт данных требует явной пометки полей, подлежащих сериализации, с помощью атрибута [DataMember]. Этот атрибут можно применять как к полям, приватным или общедоступным, так и к свойствам.

```
[DataContract(Namespace="http://www.thinktecture.com/SampleServices/2008"]
public class RoomReservation
{
  [DataMember] public string Room { get; set; }
  [DataMember] public DateTime StartDate { get; set; }
  [DataMember] public DateTime EndDate { get; set; }
  [DataMember] public string ContactName { get; set; }
  [DataMember] public string EventName { get; set; }
}
```

Для обеспечения независимости от платформы, а также возможности заменять данные новыми версиями без нарушения работы старых клиентов и служб, применение контрактов данных является наилучшим способом для указания того, какие данные должны отправляться. Можно также применять механизм сериализации XML и механизм сериализации исполняющей среды. Механизм сериализации XML обычно используется в веб-службах ASP.NET, а механизм сериализации исполняющей среды — в .NET Remoting.

В атрибуте DataMember можно указывать свойства, перечисленные в табл. 43.1.

Таблица 43.1. Свойства, задаваемые в атрибуте `DataMember`

Свойство	Описание
Name	По умолчанию сериализуемый элемент получает такое же имя, как у поля или свойства, к которому был применен атрибут [DataMember]. С помощью свойства Name это имя можно изменять.
Order	Свойство Order позволяет указать, в каком порядке должны подвергаться сериализации члены данных.
IsRequired	С помощью свойства IsRequired можно указать, что элемент должен быть обязательно получен при сериализации. Это свойство удобно использовать для управления версиями. При добавлении членов в существующий контракт, он нарушаться не будет, поскольку по умолчанию поля являются необязательными (IsRequired=false). Установив свойство IsRequired в true, существующий контракт можно нарушить.
EmitDefaultValue	Свойство EmitDefaultValue позволяет указать, должен ли член подвергаться сериализации в случае, если он имеет значение по умолчанию. Если это свойство установлено в true, то член, который имеет значение, используемое по умолчанию для его типа, сериализироваться не будет.

Управление версиями

При создании новой версии контракта данных нужно обращать внимание на то, какого рода изменения привносятся, и поступать соответствующим образом, если контракт должны поддерживать как старые, так и новые клиенты и службы.

При определении контракта с помощью свойства Namespace в атрибут DataContractAttribute потребуется добавить информацию о пространстве имен XML. В случае создания новой версии контракта данных, нарушающей совместимость, это пространство имен следует изменить.

Если добавляются лишь необязательные члены, контракт не нарушается, и такое изменение считается совместимым. Старые клиенты все равно смогут отправлять сообщения новой службе, так как дополнительные данные не являются обязательными. Новые клиенты смогут отправлять сообщения старой службе, потому что в старой службе дополнительные данные будут просто игнорироваться.

Удаление полей или добавление обязательных полей приводит к нарушению контракта. Поэтому в таком случае должно также изменяться и пространство имен XML. Название пространства имен может включать в себя год и месяц, например: `http://thinktecture.com/SampleServices/2009/10`. Каждый раз, при внесении нарушающего контракт изменения понадобится соответствующим образом изменять пространство имен, например, заменяя в нем год и месяц текущими значениями.

Контракт службы

Контракт службы определяет операции, которые может выполнять служба. Для определения самого контракта службы применяется атрибут `ServiceContract` с интерфейсами или классами. Предлагаемые службой методы снабжаются атрибутом `OperationContract`, как можно увидеть на примере интерфейса `IRoomService`:

```
[ServiceContract]
public interface IRoomService
{
  [OperationContract]
  bool ReserveRoom(RoomReservation roomReservation);
}
```

Свойства, доступные для установки в атрибуте `ServiceContract`, перечислены в табл. 43.2.

Свойства, которые можно задавать в атрибуте `OperationContract`, перечислены в табл. 43.3

Таблица 43.2. Свойства, задаваемые в атрибуте `ServiceContract`

Свойство	Описание
`ConfigurationName`	Это свойство определяет имя конфигурации службы в конфигурационном файле.
`CallbackContract`	При использовании службы для дуплексного обмена сообщениями свойство `CallbackContract` определяет контракт, который должен быть реализован на стороне клиента.
`Name`	Свойство `Name` определяет имя, которое должно использоваться для элемента `<portType>` в WSDL.
`Namespace`	Свойство `NameSpace` определяет пространство имен XML, которое должно использоваться для элемента `<portType>` в WSDL.
`SessionMode`	Свойство `SessionMode` указывает, требуются ли сеансы для выполнения определенных в данном контракте операций. Его возможными значениями являются `Allowed`, `NotAllowed` и `Required`; все они определены в перечислении `SessionMode`.
`ProtectionLevel`	Свойство `ProtectionLevel` указывает, должна ли при привязке поддерживаться защита взаимодействия. Значения, которые могут устанавливаться для этого свойства, определены в перечислении `ProtectionLevel` и выглядят следующим образом: `None`, `Sign` и `EncryptAndSign`.

Таблица 43.3. Свойства, задаваемые в атрибуте `OperationContract`

Свойство	Описание
`Action`	В WCF свойство `Action` запроса SOAP используется для его отображения на соответствующий метод. По умолчанию значением для `Action` служит сочетание названия пространства имен XML с именем контракта и именем операции. В случае если сообщение является ответным, в строку `Action` добавляется еще и слово `Response`. Устанавливая свойство `Action`, можно переопределить значение `Action`. Если присвоить этому свойству значение `"*"`, операция службы будет отвечать за обработку всех сообщений.
`ReplyAction`	В то время как свойство `Action` позволяет задавать имя `Action` для входящего запроса SOAP, свойство `ReplyAction` позволяет указывать его для ответного сообщения.
`AsyncPattern`	Если операция реализована за счет применения асинхронного шаблона, свойство `AsyncPattern` должно быть установлено в `true`. Асинхронный шаблон рассматривался в главе 20.
`IsInitiating` `IsTerminating`	Если контракт состоит из последовательности операций, начальной операции в этой последовательности необходимо присвоить свойство `IsInitiating`, а последней — свойство `IsTerminating`. Начальная операция будет запускать новый сеанс, а завершающая — закрывать его.
`IsOneWay`	В случае установки свойства `IsOneWay` клиент не будет дожидаться ответного сообщения. Инициаторы однонаправленных операций не имеют никакого прямого способа обнаруживать сбой после отправки сообщения с запросом.
`Name`	По умолчанию имя операции совпадает с именем метода, которому назначается контракт операции. С помощью свойства `Name` это имя можно изменить.
`ProtectionLevel`	С помощью свойства `ProtectionLevel` можно указать, должны ли сообщения подписываться либо подписываться и шифроваться.

С помощью атрибута `[DeliveryRequirements]` в контракте службы можно определить требования, которые должны предъявляться к службе относительно транспортировки. Свойство `RequireOrderedDelivery` позволяет указать, что отправляемые сообщения должны поступать в том же порядке, а свойство `QueuedDeliveryRequirements` — что сообщения должны отправляться в отключенном (disconnected) режиме, например, с применением технологии Message Queuing (см. главу 46).

Контракт сообщений

Контракт сообщений применяется, когда необходимо иметь полный контроль над сообщениями SOAP. В контракте сообщений можно указывать, какая часть сообщения должна относиться к заголовку SOAP, а какая — к телу SOAP. Ниже показан пример определения контракта сообщений для класса `ProcessPersonRequestMessage`. Здесь видно, что контракт сообщений задается с помощью атрибута `[MessageContract]`. Заголовок и тело сообщения SOAP специфицируются атрибутами `[MessageHeader]` и `[MessageBodyMember]`. Задавая свойство `Position`, можно определить порядок следования элементов в теле сообщения. Кроме того, можно задать желаемый уровень защиты для полей заголовка и тела.

```
[MessageContract]
public class ProcessPersonRequestMessage
{
  [MessageHeader]
  public int employeeId;

  [MessageBodyMember(Position=0)]
  public Person person;
}
```

Класс `ProcessPersonRequestMessage` используется вместе с контрактом службы, который определен с помощью интерфейса `IProcessPerson`:

```
[ServiceContract]
public interface IProcessPerson
{
  [OperationContract]
  public PersonResponseMessage ProcessPerson(
       ProcessPersonRequestMessage message);
}
```

Еще одним важным контрактом для служб WCF является контракт ошибок. Об этом контракте более подробно рассказывается в разделе "Обработка ошибок" позже в главе.

Реализация службы

Реализация службы может обозначаться атрибутом `ServiceBehavior`, как показано ниже на примере класса `RoomReservationService`:

```
[ServiceBehavior]
public class RoomReservationService: IRoomService
{
  public bool ReserveRoom(RoomReservation roomReservation)
  {
    // Реализация
  }
}
```

Атрибут `ServiceBehavior` применяется для описания поведения, которое должно предоставляться службами WCF для перехвата кода, отвечающего за реализацию требуемой функциональности. Свойства, которые могут задаваться в этом атрибуте, перечислены в табл. 43.4.

Таблица 43.4. Свойства, задаваемые в атрибуте `ServiceBehavior`

Свойство	Описание
`TransactionAutoCompleteOnSessionClose`	Это свойство позволяет указать, что при завершении текущего сеанса без ошибок фиксация транзакции должна производиться автоматически. Оно похоже на атрибут `[AutoComplete]`, который более подробно рассматривается в главе 55.
`TransactionIsolationLevel`	Это свойство позволяет задать уровень изоляции транзакции внутри службы и может принимать одно из значений перечисления `TransactionIsolationLevel`. Более подробно об уровнях изоляции транзакций рассказывалось в главе 23.

Продолжение табл. 43.4

Свойство	Описание
`ReleaseServiceInstanceOn TransactionComplete`	Это свойство позволяет указать, что при завершении транзакции экземпляр службы должен освобождаться и становиться доступным для повторного использования.
`AutomaticSessionShutdown`	Это свойство позволяет указать, должен ли закрываться сеанс при закрытии клиентом соединения. В случае установки для него значения `false` сеанс закрываться не будет. По умолчанию сеанс всегда закрывается.
`InstanceContextMode`	С помощью свойства `InstanceContextMode` можно указать, должны ли использоваться объекты с состоянием или без него. По умолчанию применяется вариант `InstanceContextMode.PerCall`, который определяет, что при каждом вызове метода должен создаваться новый объект. Другими возможными вариантами являются `PerSession` и `Single`. При любом из этих двух вариантов используются объекты с состоянием. Однако в случае `PerSession` для каждого клиента создается новый объект, а в случае `Single` — один и тот же объект используется среди множества клиентов.
`ConcurrencyMode`	Поскольку объекты с состоянием могут использоваться во множестве клиентов (или во множестве потоков одного и того же клиента), в случае их применения должно быть уделено внимание деталям, касающимся получения к ним одновременного доступа. Установка свойства `ConcurrencyMode` в `Multiple` разрешает доступ к объекту множеству потоков одновременно, при этом разработчик должен позаботиться о синхронизации доступа. Установка свойства `ConcurrencyMode` в `Single` разрешает доступ к объекту одновременно только одному потоку; в этом случае обеспечивать синхронизацию не понадобится, но нужно иметь в виду, что при большом количестве клиентов могут возникать проблемы масштабирования. Если установить это свойство в `Reentrant`, доступ к объекту будет разрешен только потоку, который происходит из вызывающего кода. Для объектов без состояния устанавливать такое значение не имеет смысла, поскольку в их случае новые объекты создаются при каждом вызове метода и потому никакого разделения состояния не происходит.
`UseSynchronizationContext`	В Windows Forms и WPF члены элементов управления могут вызываться только из потока-создателя. Если служба размещена в приложении Windows и ее методы обращаются к членам элементов управления, для свойства `UseSynchronizationContext` следует установить значение `true`. Тогда служба будет выполняться в потоке, на который указывает `SynchronizationContext`.

Окончание табл. 43.4

Свойство	Описание
`IncludeExceptionDetail InFaults`	В .NET ошибки появляются в виде исключений. В SOAP определено, что в случае возникновения проблемы на сервере клиенту возвращается ошибка SOAP. Из соображений безопасности возврат клиенту деталей возникающих на сервере исключений является нежелательным. Поэтому по умолчанию все исключения преобразуются в неизвестную ошибку. Для возврата информации о конкретной ошибке необходимо генерировать исключение типа `FaultException`. Для целей отладки очень полезна реальная информации об исключениях. В этом случае свойство `IncludeExceptionDetailInFaults` может быть установлено в `true`. В результате будет генерироваться объект `FaultException<TDetail>`, содержащий исходное исключение со всеми касающимися его деталями.
`MaxItemsInObjectGraph`	Свойство `MaxItemsInObjectGraph` позволяет ограничить количество объектов, которые должны подвергаться сериализации. Количество, установленное по умолчанию, может оказаться слишком маленьким для сериализации дерева объектов.
`ValidateMustUnderstand`	Установка свойства `ValidateMustUnderstand` в `true` означает, что понимание заголовков SOAP является обязательным (это поведение, принятое по умолчанию).

Чтобы продемонстрировать поведение службы, ниже в интерфейсе `IStateService` определен контракт службы с двумя операциями для установки и получения состояния. Для контракта службы, обладающей состоянием, должен использоваться сеанс. Именно поэтому свойство `SessionMode` в контракте службы устанавливается в `SessionMode.Required`. Также в контракте этой службы определяются методы для инициации и закрытия сеанса за счет применения свойств `IsInitiating` и `IsTerminating` к контракту операций.

```
[ServiceContract(SessionMode=SessionMode.Required)]
public interface IStateService
{
  [OperationContract(IsInitiating=true)]
  void Init(int i);

  [OperationContract]
  void SetState(int i);

  [OperationContract]
  int GetState();

  [OperationContract(IsTerminating=true)]
  void Close();
}
```

Фрагмент кода ***StateServiceSample\IStateService.cs***

Контракт этой службы реализуется в классе `StateService`. В реализации указано значение `InstanceContextMode.PerSession` для сохранения состояния с экземпляром.

```
[ServiceBehavior(InstanceContextMode=InstanceContextMode.PerSession)]
public class StateService: IStateService
{
```

```
  int i = 0;
  public void Init(int i)
  {
    this.i = i;
  }
  public void SetState(int i)
  {
    this.i = i;
  }
  public int GetState()
  {
    return i;
  }
  public void Close()
  {
  }
}
```

Фрагмент кода ***StateServiceSample\StateService.cs***

Теперь необходимо определить привязку к адресу и протоколу. Здесь конечной точке службы присваивается `basicHttpBinding`:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service behaviorConfiguration="StateServiceSample.Service1Behavior"
       name="Wrox.ProCSharp.WCF.StateService">
       <endpoint address="" binding="basicHttpBinding"
                 bindingConfiguration=""
                 contract="Wrox.ProCSharp.WCF.IStateService">
       </endpoint>
       <endpoint address="mex" binding="mexHttpBinding"
                contract="IMetadataExchange" />
        <host>
          <baseAddresses>
            <add baseAddress="http://localhost:8731/Design_Time_Addresses/
StateServiceSample/Service1/" />
          </baseAddresses>
        </host>
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
        <behavior name="StateServiceSample.Service1Behavior">
          <serviceMetadata httpGetEnabled="True"/>
          <serviceDebug includeExceptionDetailInFaults="False" />
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

Фрагмент кода ***StateServiceSample\App.config***

После запуска хоста службы с такой определенной конфигурацией, будет сгенерировано исключение `InvalidOperationException`. Оно будет сопровождаться сообщением об ошибке вида “Contract requires Session, but Binding ‘BasicHttpBinding’ doesn’t support it or isn’t configured properly to support it” (“Контракт требует использования сеанса, но привязка `BasicHttpBinding` не поддерживает его или некорректно сконфигурирована для его поддержки”).

Не все привязки поддерживают все службы. Поскольку контракт службы требует использования сеанса согласно атрибуту `[ServiceContract(SessionMode=SessionMode.Required)]`, хосту не удается справиться с задачей из-за того, что сконфигурированная привязка не поддерживает сеансы. В случае замены привязки в конфигурации такой, которая поддерживает сеансы (например, `wsHttpBinding`), сервер запустится нормально:

```
<endpoint address="" binding="wsHttpBinding" bindingConfiguration=""
    contract="Wrox.ProCSharp.WCF.IStateService">
</endpoint>
```

*Фрагмент кода **StateServiceSample\App.config***

Создание клиента программным образом

Теперь можно переходить к созданию клиентского приложения. В предыдущем примере клиентское приложение создавалось за счет добавления ссылки на службу. Однако вместо того чтобы добавлять такую ссылку, можно напрямую получить доступ к содержащей контракт интерфейса сборке и воспользоваться классом `ChannelFactory<TChannel>` для создания экземпляра канала, подключаемого к службе.

Конструктор класса `ChannelFactory<TChannel>` в качестве параметров принимает конфигурацию привязки и адрес конечной точки. Конфигурация привязки должна обязательно быть совместима с той, что была определена для хоста службы, а адрес конечной точки — определяться с помощью класса `EndpointAddress` и ссылаться на URI-адрес работающей службы.

Метод `CreateChannel()` позволяет создавать канал для подключения к службе. После него могут вызываться любые необходимые методы службы. Экземпляр службы будет хранить свое состояние до тех пор, пока не будет вызван метод `Close()`, которому здесь было назначено поведение операции `IsTerminating`.

```
using System;
using System.ServiceModel;

namespace Wrox.ProCSharp.WCF
{
  class Program
  {
    static void Main()
    {
      var binding = new WSHttpBinding();
      var address = new EndpointAddress("http://localhost:8731/" +
          "Design_Time_Addresses/StateServiceSample/Service1/");
      var factory = new ChannelFactory<IStateService>(binding, address);

      IStateService channel = factory.CreateChannel();
      channel.Init(1);
      Console.WriteLine(channel.GetState());
      channel.SetState(2);

      Console.WriteLine(channel.GetState());
      channel.Close();
      factory.Close();
    }
  }
}
```

*Фрагмент кода **StateClient\Program.cs***

При реализации службы к ее методам с помощью атрибута `OperationBehavior` можно применять свойства, перечисленные в табл. 43.5.

Таблица 43.5. Свойства, задаваемые в атрибуте `OperationBehavior`

Свойства	Описание
`AutoDisposeParameters`	По умолчанию все освобождаемые (disposable) параметры удаляются из памяти автоматически. Если это не нужно, установите свойство `AutoDisposeParameters` в `false`. В таком случае за удаление параметров будет отвечать отправитель.
`Impersonation`	С помощью свойства `Impersonation` вызывающий код может заимствовать права, и тогда метод будет выполняться с использованием его идентификационных данных.
`ReleaseInstanceMode`	Свойство `InstanceContextMode` позволяет определять время жизни экземпляра объекта в атрибуте `ServiceBehavior`, которое может быть переопределено в атрибуте `OperationBehavior` за счет указания желаемой операции. Свойство `ReleaseInstanceMode` позволяет задавать желаемый режим освобождения экземпляра с использованием одного из значений перечисления `ReleaseInstanceMode`. Значение `None` предусматривает применение настроек, указанных в `InstanceContextMode`, а значения `BeforeCall`, `AfterCall` и `BeforeAndAfterCall` позволяют указать, когда должна повторяться операция, указанная в `OperationBehavior`.
`TransactionScopeRequired`	С помощью свойства `TransactionScopeRequired` можно указать, должна ли вместе с операцией использоваться транзакция. Если транзакция требуется, и вызывающая сторона уже ее запустила, будет использоваться эта транзакция, а если вызывающая сторона никаких транзакций не запускала, будет создана новая транзакция.
`TransactionAutoComplete`	Свойство `TransactionAutoComplete` позволяет указать, должна ли транзакция завершаться автоматически. Если оно установлено в `true`, то в случае выдачи исключения транзакция будет прерываться, а в случае если она является корневой и никакого исключения не генерируется, транзакция будет фиксироваться.

Обработка ошибок

По умолчанию детальная информация об исключениях, которые возникают в службе, клиентскому приложению не возвращается. Такое поведение связано с безопасностью. Никому не нужно, чтобы при использовании его службы третья сторона могла получать детальные сведения о генерируемых в ней исключениях. Вместо этого обычно требуется, чтобы все исключения фиксировались в самой службе (этого можно добиться посредством трассировки и регистрации событий в журналах), и чтобы вызывающей стороне возвращались только содержащие полезную информацию сообщения об ошибках.

Для возврата ошибки SOAP можно сгенерировать исключение `FaultException`. При этом создается нетипизированная ошибка SOAP. Однако более предпочтительным вариантом является генерация строго типизированной ошибки SOAP.

Информация, которая должна передаваться вместе со строго типизированной ошибкой SOAP, определена в контракте данных, как показано ниже на примере класса `StateFault`:

```
[DataContract]
public class StateFault
{
  [DataMember]
  public int BadState { get; set; }
}
```

*Фрагмент кода **StateServiceSample\IStateService.cs***

Тип ошибки SOAP определяется в контракте операций с помощью атрибута `FaultContractAttribute`:

```
[FaultContract(typeof(StateFault))]
[OperationContract]
void SetState(int i);
```

В реализации генерируется исключение `FaultException<TDetail>`. В конструкторе может назначаться новый объект `TDetail`, которым в рассматриваемом примере является `StateFault`. Помимо этого в конструкторе экземпляру `FaultReasonText` может быть присвоена информация об ошибке. Класс `FaultReasonText` поддерживает возможность указания информации об ошибках на множестве различных языков.

```
public void SetState(int i)
{
  if (i == -1)
  {
    FaultReasonText[] text = new FaultReasonText[2];
    text[0] = new FaultReasonText("Sample Error",
              new CultureInfo("en"));
    text[1] = new FaultReasonText("Beispiel Fehler",
              new CultureInfo("de"));
    FaultReason reason = new FaultReason(text);
    throw new FaultException<StateFault>(
          new StateFault() { BadState = i }, reason);
  }
  else
  {
    this.i = i;
  }
}
```

*Фрагмент кода **StateServiceSample\StateService.cs***

В клиентском приложении исключения типа `FaultException<StateFault>` могут перехватываться. Причина выдачи исключения определяется в свойстве `Message`, а доступ к `StateFault` получается с помощью свойства `Detail`:

```
try
{
  channel.SetState(-1);
}
catch (FaultException<StateFault> ex)
{
  Console.WriteLine(ex.Message);
  StateFault detail = ex.Detail;
  Console.WriteLine(detail.BadState);
}
```

*Фрагмент кода **StateServiceClient\Program.cs***

Помимо строго типизированных ошибок SOAP в клиентском приложении могут также перехватываться и исключения базовых по отношению к `FaultException<Detail>`

классов `FaultException` и `CommunicationException`. Перехватывая исключения `CommunicationException`, можно перехватывать и другие исключения, связанные с коммуникациями WCF.

Привязка

Привязка позволяет описывать то, каким образом служба должна осуществлять взаимодействие. В привязке можно задавать следующие вещи:

- транспортный протокол;
- безопасность;
- формат кодировки;
- поток транзакций;
- надежность;
- изменение формы;
- обновление транспорта.

Привязка состоит из множества элементов, которые описывают все предъявляемые к привязке требования. Можно создавать как собственную привязку, так и применять какую-то из предопределенных привязок, которые перечислены в табл. 43.6.

Таблица 43.6. Стандартные привязки

Стандартная привязка	Описание
`BasicHttpBinding`	Привязка `BasicHttpBinding` предназначена для предоставления самых широких функциональных возможностей по взаимодействию для веб-служб первого поколения. Она предусматривает применение таких транспортных протоколов, как HTTP и HTTPS; доступны только средства безопасности, поддерживаемые этими протоколами.
`WSHttpBinding`	Привязка `WSHttpBinding` предназначена для веб-служб следующего поколения, т.е. платформ, которые реализуют расширения SOAP для обеспечения безопасности, надежности и возможности выполнения транзакций. Она использует транспортные протоколы HTTP и HTTPS, а также позволяет применять спецификацию WS-Security для обеспечения безопасности, спецификации WS-Coordination, WS-AtomicTransaction и WS-BusinessActivity для поддержки транзакций, и реализацию WS-ReliableMessaging для обеспечения надежного обмена сообщениями. Кроме того, для отправки вложений поддерживается спецификация с возможностью выполнения кодирования по протоколу MTOM (Message Transmission Optimization Protocol — протокол оптимизации передачи сообщений). Дополнительные сведения по всем этим спецификациям стандарта WS-* доступны по адресу `http://www.oasis-open.org`.
`WS2007HttpBinding`	Привязка `WS2007HttpBinding` унаследована от базового класса `WSHttpBinding` и дополнительно поддерживает возможность использования спецификаций по обеспечению безопасности, надежности и выполнению транзакций, которые были определены группой OASIS (Organization for the Advancement of Structured Information Standards — организация по продвижению стандартов для структурированной информации).

Окончание табл. 43.6

Стандартная привязка	Описание
`WSHttpContextBinding`	Привязка `WSHttpContextBinding` унаследована от базового класса `WSHttpBinding` и дополнительно поддерживает возможность использования контекста без применения механизма cookie-наборов. Она включает в себя дополнительный элемент `ContextBindingElement` для обмена контекстной информацией.
`WebHttpBinding`	Эта привязка применяется для служб, доступ к которым предоставляется посредством HTTP-запросов, а не запросов SOAP. Это полезно для сценарных клиентов, например, ASP.NET AJAX.
`WSFederationHttpBinding`	`WSFederationHttpBinding` представляет собой безопасную и способную к взаимодействию привязку, которая поддерживает разделение идентификационных данных среди множества систем для выполнения аутентификации и авторизации.
`WSDualHttpBinding`	Привязка `WSDualHttpBinding`, в отличие от `WSHttpBinding`, поддерживает возможность дуплексного обмена сообщениями.
`NetTcpBinding`	Все стандартные привязки, имена которых начинаются с `Net`, используют двоичное кодирование для коммуникаций между приложениями .NET. Такое кодирование является более быстрым, чем текстовое кодирование, применяемое в привязках `WSxxx`. Привязка `NetTcpBinding` использует протокол TCP/IP.
`NetTcpContextBinding`	Подобно `WSHttpContextBinding`, привязка `NetTcpContextBinding` включает дополнительный элемент `ContextBindingElement` для обмена контекстом с заголовком SOAP.
`NetPeerTcpBinding`	Привязка `NetPeerTcpBinding` предназначена для одноранговых коммуникаций.
`NetNamedPipeBinding`	Привязка `NetNamedPipeBinding` оптимизирована для коммуникаций между различными процессами в одной и той же системе.
`NetMsmqBinding`	Привязка `NetMsmqBinding` привносит в WCF возможность использования очередей сообщений, т.е. сообщения отправляются в очередь сообщений.
`MsmqIntegrationBinding`	Привязка `MsmqIntegrationBinding` предназначена для применения в существующих приложениях, в которых используются очереди сообщений. В отличие от нее привязка `NetMsmqBinding` требует приложений WCF на стороне клиента и на стороне сервера.
`CustomBinding`	С помощью привязки `CustomBinding` можно должным образом настроить требования к транспортному протоколу и безопасности.

Каждая привязка поддерживает свою функциональность. Привязки, имена которых начинаются с `WS`, имеют независимую от платформы природу и поддерживают всю функциональность, которая перечислена в спецификациях веб-служб. Привязки, имена которых начинаются с `Net`, используют двоичное форматирование для обеспечения высокоскоростного взаимодействия между приложениями .NET. К числу других функциональных средств относятся сеансы, надежные сеансы, транзакции и дуплексное взаимодействие; в табл. 43.7 перечислены привязки, поддерживающие такую функциональность.

Таблица 43.7. Средства и поддерживающие их привязки

Средство	Привязки
Сеансы	WSHttpBinding, WSDualHttpBinding, WsFederationHttpBinding, NetTcpBinding, NetNamedPipeBinding
Надежные сеансы	WSHttpBinding, WSDualHttpBinding, WsFederationHttpBinding, NetTcpBinding
Транзакции	WSHttpBinding, WSDualHttpBinding, WSFederationHttpBinding, NetTcpBinding, NetNamedPipeBinding, NetMsmqBinding, MsmqIntegrationBinding
Дуплексное взаимодействие	WsDualHttpBinding, NetTcpBinding, NetNamedPipeBinding, NetPeerTcpBinding

Наряду с привязкой в службе должна быть определена конечная точка, которая зависит от контракта, адреса службы и привязки. В следующем примере кода создается экземпляр объекта ServiceHost, после чего его адрес http://localhost:8080/RoomReservation, экземпляр WsHttpBinding и контракт добавляются в конечную точку службы.

```
static ServiceHost host;

static void StartService()
{
  var baseAddress = new Uri("http://localhost:8080/RoomReservation");
  host = new ServiceHost(
    typeof(RoomReservationService));

  var binding1 = new WSHttpBinding();
  host.AddServiceEndpoint(typeof(IRoomService), binding1, baseAddress);
  host.Open();
}
```

Привязку можно определять не только программно, но и в конфигурационном файле приложения. Конфигурация для WCF размещена внутри элемента <system.serviceModel>. В элементе <service> указываются предлагаемые службы. Как и в коде, здесь для службы необходимо указать конечную точку, в которую включить информацию об адресе, привязке и контракте. По умолчанию конфигурация привязки wsHttpBinding изменяется за счет добавления XML-атрибута bindingConfiguration, который ссылается на конфигурацию привязки wsHttpConfig1. Именно такую конфигурацию привязки можно найти внутри раздела <binding> и именно ее следует использовать для изменения конфигурации wsHttpBinding так, чтобы она поддерживала reliableSession.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name="Wrox.ProCSharp.WCF.RoomReservationService">
       <endpoint address=" http://localhost:8080/RoomReservation"
          contract="Wrox.ProCSharp.WCF.IRoomService"
         binding="wsHttpBinding" bindingConfiguration="wsHttpBinding" />
      </service>
    </services>
    <bindings>
      <wsHttpBinding>
        <binding name="wsHttpBinding">
          <reliableSession enabled="true" />
        </binding>
```

```
      </wsHttpBinding>
    </bindings>
  </system.serviceModel>
</configuration>
```

Хостинг

Технология WCF является очень гибкой в том, что касается выбора хоста для запуска службы. В роли хоста может выступать служба Windows, приложение COM+, WAS (Windows Activation Services — службы активации Windows) или IIS, приложение Windows и даже простое консольное приложение. При создании специального хоста с помощью Windows Forms или WPF можно получить одноранговое решение.

Специальный хост

Начнем со специального хоста. В предлагаемом здесь примере демонстрируется создание специального хоста для службы внутри консольного приложения, однако в других специальных хостах, таких как службы или приложения Windows, служба может программироваться тем же самым образом.

В методе `Main()` создается экземпляр `ServiceHost`. После его создания производится чтение конфигурационного файла приложения для определения привязок. Как было показано ранее, привязки могут также определяться программно. Затем вызывается метод `Open()` класса `ServiceHost`, чтобы служба могла принимать клиентские вызовы. В консольном приложении необходимо соблюдать осторожность и не закрывать главный поток до тех пор, пока не потребуется закрыть службу. Здесь пользователю предлагается нажать клавишу <Enter> для выхода из службы. После этого вызывается метод `Close()` для действительного завершения работы службы.

```
using System;
using System.ServiceModel;
public class Program
{
  public static void Main()
  {
    using (var serviceHost = new ServiceHost())
    {
      serviceHost.Open();
      Console.WriteLine("Служба запущена. Для завершения нажмите <Enter>");
      Console.ReadLine();
      serviceHost.Close();
    }
  }
}
```

Для прерывания работы хоста службы можно вызвать метод `Abort()` класса `ServiceHost`, а для получения информации о текущем состоянии — воспользоваться свойством `State`, которое возвращает одно из значений, определенных в перечислении `CommunicationState`. Возможные значения выглядят следующим образом: `Created`, `Opening`, `Opened`, `Closing`, `Closed` и `Faulted`.

Если в коде службы, запущенной из приложения Windows Forms или WPF, присутствуют обращения к методам элементов управления Windows, нужно обязательно позаботиться о том, чтобы доступ к методам и свойствам этих элементов управления был разрешен только создавшему их потоку. В WCF это можно сделать, установив соответствующее значение для свойства `UseSynchronizatonContext` в атрибуте `[ServiceBehavior]`.

Хост WAS

Использование компонента WAS (Windows Activation Services — службы активации Windows) в качестве хоста позволяет получить такие средства рабочего процесса WAS, как автоматическая активация службы, мониторинг работоспособности и повторное использование процессов.

Чтобы использовать WAS в качестве хоста, необходимо создать веб-сайт и файл `.svc` с объявлением `ServiceHost`, включающим язык и имя класса службы. В предлагаемом примере кода применяется класс `Service1`. Вдобавок нужно указать файл, в котором содержится класс службы. Этот класс реализуется таким же образом, как показывалось ранее при определении библиотеки службы WCF.

```
<%@ ServiceHost language="C#" Service="Service1" CodeBehind="Service1.svc.cs" %>
```

Если используется библиотека службы WCF, которая должна быть доступна из хоста WAS, можно создать `.svc`, содержащий лишь ссылку на этот класс:

```
<%@ ServiceHost Service="Wrox.ProCSharp.WCF.Services.RoomReservationService" %>
```

Начиная с выхода ОС Windows Vista и Windows Server 2008, компонент WAS позволяет определять привязки .NET TCP и Message Queue. В случае работы с предыдущей версией — IIS 6 или IIS 5.1 из Windows Server 2003 и Windows XP — активация из файла `.svc` может осуществляться только с помощью привязки HTTP.

Службу WCF можно также добавить в компоненты Enterprise Service. Более подробно об этом будет рассказываться в главе 51.

Предварительно сконфигурированные классы хостов

Для сокращения работы, связанной с конфигурированием, в WCF предлагаются классы хостов с предварительно сконфигурированными привязками. Одним из примеров таких классов является `WebServiceHost`, который находится в сборке `System.ServiceModel.Web` из пространства имен `System.ServiceModel`. Этот класс предусматривает создание стандартной конечной точки для базовых адресов HTTP и HTTPS, если не определено никакой стандартной конечной точки с помощью `WebHttpBinding`, а также добавление поведения `WebHttpBehavior`, если не определено другое поведение. Благодаря такому поведению, простые HTTP-операции `GET`, `POST` , `PUT` и `DELETE` (задаваемые с помощью атрибута `WebInvoke`) могут производиться без дополнительной настройки.


```
Uri baseAddress = new Uri("http://localhost:8000/RoomReservation");
var host = new WebServiceHost(typeof(RoomReservationService), baseAddress);
host.Open();

Console.WriteLine("служба запущена");
Console.WriteLine("Для завершения нажмите <Enter>...");
Console.ReadLine();
if (host.State == CommunicationState.Opened)
host.Close();
```

Фрагмент кода ***RoomReservationWebServiceHost\Program.cs***

Чтобы использовать HTTP-запрос `GET` для получения заявок на резервирование комнат, методу `GetRoomReservation` необходим атрибут `WebGet`, отображающий параметры метода на входные данные из запроса `GET`. В следующем коде в этом атрибуте определяется шаблон `UriTemplate`, который требует, чтобы `Reservations` добавлялось к базовому адресу и сопровождалось параметрами `From` и `To`. Эти параметры, в свою очередь, отображаются на переменные `fromDate` и `toDate`.

```
[WebGet(UriTemplate="Reservations?From={fromDate} & To={toDate}")]
public RoomReservation[] GetRoomReservations(DateTime fromDate, DateTime toDate)
{
  var data = new RoomReservationData();
  return data.GetReservations(fromDate, toDate);
}
```

Фрагмент кода ***RoomReservationService\RoomReservationService.cs***

Теперь служба может вызываться с помощью простого запроса, показанного ниже, и будет возвращать информацию обо всех заявках на резервирование, которые были сделаны за указанный промежуток времени:

```
http://localhost:8000/RoomReservation/Reservations?From=2010/1/1&To=2010/8/1
```

Еще одним доступным классом с предварительно сконфигурированными функциональными возможностями является `System.Data.Services.DataServiceHost`. *Этот класс унаследован от* `WebServiceHost` *и предлагает службы данных, которые более подробно рассматривались в главе 32.*

Клиенты

Клиентскому приложению для получения доступа к службе необходим прокси-класс. Создается такой прокси-класс для клиента одним из трех способов.

- **В диалоговом окне Add Service Reference из Visual Studio.** Это окно позволяет создавать прокси-класс из метаданных службы.
- **С помощью утилиты ServiceModel Metadata Utility (`Svcutil.exe`).** Эта утилита позволяет создавать прокси-класс за счет считывания метаданных из службы.
- **С помощью класса `ChannelFactory`.** Этот класс используется прокси-классом, сгенерированным с помощью утилиты `Svcutil.exe`; однако он может также применяется для создания прокси-класса программным образом.

При добавлении ссылки на службу в окне Add Service Reference необходим доступ к документу WSDL. Этот документ WSDL создается конечной точкой MEX, которая должна быть сконфигурирована вместе со службой. В показанной ниже конфигурации в конечной точке с относительным адресом `mex` используется `mexHttpBinding` и реализуется контракт `IMetadataExchange`. Для доступа к метаданным с помощью HTTP-запроса `GET` конфигурируется `behaviorConfiguration` со значением `MexServiceBehavior`.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service behaviorConfiguration="MexServiceBehavior"
        name="Wrox.ProCSharp.WCF.RoomReservationService">
        <endpoint address="Test" binding="wsHttpBinding"
           contract="Wrox.ProCSharp.WCF.IRoomService" />
        <endpoint address="mex" binding="mexHttpBinding"
           contract="IMetadataExchange" />
        <host>
          <baseAddresses>
            <add baseAddress=
             "http://localhost:8732/Design_Time_Addresses/RoomReservationService/" />
          <baseAddresses>
        </host>
      </service>
    </services>
```

```
    <behaviors>
      <serviceBehaviors>
        <behavior name="MexServiceBehavior">
          <!-- Во избежание раскрытия информации о метаданных, перед развертыванием
          идущее ниже значение должно быть установлено в false, а идущую выше
          конечная точка метаданных - удалена -->
          <serviceMetadata httpGetEnabled="True"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

Как и средству Add Service Reference в Visual Studio, утилите `Svcutil.exe` для создания прокси-класса требуются метаданные. Утилита `Svcutil.exe` может создавать прокси-данные из конечной точки метаданных MEX, метаданных сборки и документации WSDL или XSD:

```
svcutil http://localhost:8080/RoomReservation?wsdl /language:C# /out:proxy.cs
svcutil CourseRegistration.dll
svcutil CourseRegistration.wsdl CourseRegistration.xsd
```

После генерации прокси-класса остается лишь создать его экземпляр в клиентском коде, вызвать нужные методы и в конце вызвать метод `Close()`:

```
var client = new RoomServiceClient();
client.RegisterForCourse(roomReservation);
client.Close();
```

Генерируемый прокси-класс унаследован от базового класса `ClientBase<TChannel>`, который служит оболочкой для класса фабрики `ChannelFactory<TChannel>`. Вместо сгенерированного прокси-класса можно использовать непосредственно класс `ChannelFactory<TChannel>`. Конструктору необходимо передать привязку и адрес конечной точки, затем создать канал и вызвать методы, определенные контрактом службы. Напоследок класс фабрики должен быть закрыт.

```
var binding = new WsHttpBinding();
var address = new EndpointAddress("http://localhost:8080/RoomService");
var factory =  new ChannelFactory<IStateService>(binding, address);

IRoomService channel = factory.CreateChannel();
channel.ReserveRoom(roomReservation);
//...
factory.Close();
```

Класс `ChannelFactory<TChannel>` имеет ряд свойств и методов, которые перечислены в табл. 43.8.

Таблица 43.8. Члены класса `ChannelFactory`

Член	Описание
`Credentials`	`Credentials` представляет собой свойство только для чтения, которое позволяет получать доступ к объекту `ClientCredentials`, назначаемому каналу для прохождения аутентификации в службе. Учетные данные могут быть установлены вместе с конечной точкой.
`Endpoint`	`Endpoint` представляет собой свойство только для чтения, которое позволяет получать доступ к ассоциируемому с каналом экземпляру `ServiceEndpoint`. Конечная точка может назначаться в конструкторе.

Окончание табл. 43.8

Член	Описание
`State`	`State` представляет собой свойство типа `CommunicationState` и возвращает информацию о текущем состоянии канала. `CommunicationState` — перечисление со значениями `Created`, `Opening`, `Opened`, `Closing`, `Closed` и `Faulted`.
`Open()`	Метод `Open()` открывает канала.
`Close()`	Метод `Close()` закрывает канал.

`Opening`
`Opened`
`Closing`
`Closed`
`Faulted`
	Этим событиям можно назначить обработчики и получать уведомления об изменениях в состоянии канала. Эти события срабатывают, соответственно, перед и после открытия канала, перед и после его закрытия, а также в случае возникновения сбоя.

Дуплексные коммуникации

В следующем примере приложения показано, как установить дуплексные коммуникации между клиентом и службой. Клиент инициирует установку соединения со службой. После установки соединения служба может осуществлять обратные вызовы клиента.

Для дуплексных коммуникаций должен быть определен контракт, реализуемый в клиенте. Здесь контракт для клиента определяется с помощью интерфейса `IMyMessageCallback`. Методом, который должен реализовать клиент, является `OnCallback()`. Для операции в атрибуте `OperationContract` применяется настройка `IsOneWay=true`. Благодаря этому, служба не будет ожидать успешного вызова метода на стороне клиенте. По умолчанию экземпляр службы может вызываться только из одного потока (в чем можно убедиться, взглянув на свойство `ConcurrencyMode` в атрибуте `ServiceBehavior`, для которого по умолчанию устанавливается значение `ConcurrencyMode.Single`).

Если в реализации службы выполняется обратный вызов клиента с ожиданием ответа, то поток, получающий этот ответ, должен будет ждать получения блокировки объекта службы. Из-за того, что объект службы уже заблокирован запросом клиента, возникнет взаимоблокировка. WCF обнаружит эту взаимоблокировку и сгенерирует исключение. Чтобы избежать возникновения подобной ситуации, можно заменить значение свойства `ConcurrencyMode` на `Multiple` или `Reentrant`. Если это свойство установлено в `Multiple`, то получать доступ к экземпляру сможет множество потоков одновременно и, следовательно, потребуется реализовать собственный механизм блокировок. При установке для него значения `Reentrant` экземпляр службы останется однопоточным, но позволит повторный вход в контекст ответов на запросы обратного вызова. Вместо того чтобы изменять значение свойства `ConcurrencyMode`, можно задать в атрибуте `OperationContract` свойство `IsOneWay`. Тогда вызывающая сторона не будет дожидаться ответа. Разумеется, подобный вариант допустим только в случае, если возвращаемые значения не ожидаются.

Контракт службы определен интерфейсом `IMyMessage`. Контракт обратного вызова отображается на контракт службы за счет применения в определении контракта службы свойства `CallbackContract`:

```
public interface IMyMessageCallback
{
  [OperationContract(IsOneWay=true)]
  void OnCallback(string message);
}
[ServiceContract(CallbackContract=typeof(IMyMessageCallback))]
```

```
public interface IMyMessage
{
  [OperationContract]
  void MessageToServer(string message);
}
```

Фрагмент кода ***MessageService\IMyMessage.cs***

Классе `MessageService` реализует контракт службы `IMyMessage`. Служба выводит сообщение от клиента на консоль. Для доступа к контракту обратного вызова можно использовать класс `OperationContext`. Свойство `OperationContext.Current` возвращает экземпляр `OperationContext`, ассоциированный с текущим запросом от клиента. С помощью `OperationContext` можно получать доступ к информации о сеансе, заголовкам сообщений и свойствам сообщения, а в случае дуплексной связи — и к каналу обратного вызова. Обобщенный метод `GetCallbackChannel()` возвращает информацию о канале связи с экземпляром клиента. Этот канал затем может использоваться для отправки сообщений клиенту за счет вызова метода `OnCallBack()`, который был определен в интерфейсе обратного вызова `IMyMessageCallback`. Для демонстрации того, что также возможно использовать канал обратного вызов из службы и независимым от выполнения метода образом, создается новый поток, который получает канал обратного вызова в качестве параметра и затем использует его для отправки сообщений клиенту.

```
public class MessageService: IMyMessage
{
  public void MessageToServer(string message)
  {
    Console.WriteLine("сообщение от клиента: {0}", message);
    IMyMessageCallback callback =
       OperationContext.Current.
          GetCallbackChannel<IMyMessageCallback>();

    callback.OnCallback("сообщение от сервера");
    new Thread(ThreadCallback).Start(callback);
  }
  private void ThreadCallback(object callback)
  {
    IMyMessageCallback messageCallback = callback as IMyMessageCallback;
    for (int i = 0; i < 10; i++)
    {
      messageCallback.OnCallback("сообщение " + i.ToString());
      Thread.Sleep(1000);
    }
  }
}
```

Фрагмент кода ***MessageService\MessageService.cs***

Хост службы выглядит точно так же, как и в предыдущих примерах, и потому здесь не показан. Однако для обеспечения дуплексных коммуникаций обязательно должна быть сконфигурирована привязка, поддерживающая дуплексный канал. Одной из таких привязок является `wsDualHttpBinding`, которая и применяется в конфигурационном файле приложения.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name="Wrox.ProCSharp.WCF.MessageService">
       <endpoint contract="Wrox.ProCSharp.WCF.IMyMessage"
          binding="wsDualHttpBinding" />
```

```
        <host>
          <baseAddresses>
            <add baseAddress="http://localhost:8732/Service1" />
          </baseAddresses>
        </host>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

Фрагмент кода ***MessageService\App.config***

В клиентском приложении должен быть реализован контракт обратного вызова, как показано здесь на примере класса `ClientCallback`, реализующего интерфейс `IMyMessageCallback`:

```
class ClientCallback: IMyMessageCallback
{
  public void OnCallback(string message)
  {
    Console.WriteLine("сообщение от сервера: {0}", message);
  }
}
```

Фрагмент кода ***MessageClient\Program.cs***

В случае дуплексного канала использовать `ChannelFactory` для установки соединения со службой, как это делалось ранее, нельзя. Для создания дуплексного канала необходимо применять класс `DuplexChannelFactory`. Этот класс имеет конструктор, принимающий помимо конфигурации привязки и адреса еще один параметр. Этим параметром является `InstanceContext`, упаковывающий в оболочку один экземпляр класса `ClientCallback`. При передаче этого экземпляра фабрике служба может вызывать объект через канал. Клиенту просто должен удерживать соединение открытым. После закрытия соединения служба не сможет пересылать по нему сообщения.

```
var binding = new WSDualHttpBinding();
var address = new EndpointAddress("http://localhost:8732/Service1");

ClientCallback clientCallback = new ClientCallback();
InstanceContext context = new InstanceContext(clientCallback);

DuplexChannelFactory<IMyMessage> factory =
   new DuplexChannelFactory<IMyMessage>(context, binding, address);

IMyMessage messageChannel = factory.CreateChannel();

messageChannel.MessageToServer("From the client");
```

Дуплексные коммуникации достигаются за счет запуска хоста службы и клиентского приложения.

Резюме

В этой главе было показано, как применять Windows Communication Foundation для организации коммуникаций между клиентом и сервером. Как и веб-службы ASP.NET, WCF является независимой от платформы технологией, но предоставляет больше средств, подобных .NET Remoting, Enterprise Services и Message Queuing.

В WCF очень важную роль играют контракты, потому что именно они упрощают изоляцию разрабатываемых клиентов и служб и позволяют обеспечивать независимость от платформы. Эти контракты делятся на три типа: контракты служб, контракты данных и

контракты сообщений. Для определения поведения службы и ее операций в WCF можно использовать несколько различных атрибутов.

Также в главе рассматривалось создание клиентов из предлагаемых службой метаданных и с применением контракта с интерфейсом .NET. Было показано, какую функциональность можно получить в распоряжение в случае использования различных вариантов привязки. В WCF предлагаются привязки не только для обеспечения независимости от платформы, но и для реализации быстрых коммуникаций между приложениями .NET. Было продемонстрировано создание специальных хостов и применение хоста WAS, а также достижение дуплексных коммуникаций за счет определения интерфейса обратного вызова, применения контракта службы и реализации контракта обратного вызова в клиентском приложении.

В следующих нескольких главах будет продолжено рассмотрение возможностей WCF. Глава 44 посвящена Windows Workflow Foundation и использованию WCF для взаимодействия с экземплярами рабочих потоков. В главе 46 показано, как применять с привязками WCF автономные средства Message Queuing. В главе 47 рассматриваются возможности, доступные в WCF для синдикации, а в главе 51 — способы интеграции Enterprise Services с WCF.