2

Соглашения о службах и соглашения о данных

В ЭТОЙ ГЛАВЕ...

- Описание соглашений
- Реализация соглашений о службах
- Реализация соглашений о данных
- Использование различных преобразователей
- Реализация соглашений о сообщениях

Соглашения в целом очень важны в нашей повседневной жизни. Благодаря соглашениям каждая из сторон знает, чем окупятся ее усилия. Если вы поставили свою подпись в конце 17 страниц деловой документации или заключили устное соглашение, ожидания считаются определенными и согласованными. Соглашения должны быть четко определены и понятны и не должны оставлять свободу для интерпретации.

Ясно одно: в служебно-ориентированной среде, в которой различные субъекты (компоненты программного обеспечения) взаимодействуют друг с другом и используют различные технологии и платформы, крайне необходимо иметь четкое определение всего, что при этом происходит.

WCF (Windows Communication Foundation) использует концепцию соглашений для определения службы и его функционирования в целом, для описания передаваемых данных и, если необходимо, непосредственного определения SOAP-сообщений, которыми обмениваются клиент и служба. WCF использует WSDL и XSD для доставки метаданных службы.

В WCF есть три типа соглашений: о службах, данных и сообщениях.

Каждое из этих соглашений определяет свои характеристики. Какие из них необходимы, зависит от вашей архитектуры. Наиболее часто используется соглашение о службе, которое может использоваться почти в любом сценарии. Соглашение о службе также определяет конечную точку в цепи. Как вы знаете, конечной точкой службы является адресуемый элемент, который обычно используется клиентом для отправки и получения сообщений.

Соглашения в контексте SOA обеспечивают необходимые метаданные для связи со службой, — они описывают такие вещи, как типы данных, операции, шаблоны обмена сообщениями, а также используемый транспортный протокол. Соглашения, как правило, поставляются на стандартизированной платформе и в независимом от языка формате XML, что позволяет различным сторонам использовать и интерпретировать их определение. В WCF метаданные службы описываются в основном документом WSDL (Web Services Description Language). Более подробная информация доступна по адресу `http://www.w3.org/TR/wsdl`. Этот стандарт позволяет поставщикам программного обеспечения, таким как Microsoft, Sun и IBM, разрабатывать инструменты, помогающие разработчикам создавать и использовать эти соглашения.

Соглашения о службах

Соглашения о службах определяют функциональные возможности, которые ваш служба предлагает внешнему миру. Функциональные возможности выражаются в форме операций службы. В мире SOA вы обычно обмениваетесь набором SOAP-сообщений, и схемы этих сообщений определяются в соглашении о службе. Соглашение о службе также определяет шаблон обмена сообщениями (запрос/ответ, односторонний, дуплекс) для каждой операции службы.

Соглашения о данных

Обычно вы передаете один или несколько аргументов операциям службы и ожидаете возврата значения. Структура аргументов сообщения определяется в соглашении о данных в виде схемы XSD, которая является частью документа WSDL. Соглашение о данных определяет структуру и содержание информации, которой обмениваются клиент и служба.

Если вы используете собственные объекты и сложные типы в качестве аргументов метода или возвращаемых значений, то должны информировать выполняющуюся программу, как преобразовать эти типы в поток XML с помощью атрибута `DataContract`.

В зависимости от детализации аргументов вашего метода можно использовать только простые .NET типы как аргументы метода. В этом случае вам не нужно беспокоиться о преобразовании, потому что инфраструктура обмена сообщениями уже знает, как обрабатывать эти простые .NET типы.

Соглашения о сообщениях

Соглашение о сообщениях — это свойство, которое дает дополнительные возможности управления заголовком или телом SOAP. Соглашение о сообщениях используется для непосредственного взаимодействия с сообщением SOAP (заголовком и телом), а не только с телом метода, как и в случае, если вы используете соглашение о данных, а не о сообщении.

Соглашение и код

Разработчику .NET программного обеспечения для службы далеко не каждый день приходится писать файлы метаданных в виде WSDL, — ему, как правило, нужно создавать интерфейсы и классы, писать методы, реализовывать бизнес-логику, а затем писать тесты модулей, чтобы проверить логику. Чтобы помочь разработчику программного обеспечения создавать эти искусственные объекты метаданных из исходных кодов без коренного изменения программного кода, WCF предлагает различные инструменты, атрибуты и классы, которые более подробно будут рассмотрены далее.

При использовании WCF для реализации архитектуры SOA можно выбрать различные подходы в зависимости от требований — автоматическое создание этих документов метаданных из заготовок на основе кода. Можно использовать атрибуты .NET, параметры конфигурации, а также непосредственно предоставляемые WCF средства API для полного управления генерацией метаданных.

Но как обрабатывать метаданные, полученные от клиентов, и взаимодействовать со службой? Для завершения картины предположим, что у вас уже сформирована спецификация службы в виде документа WSDL. Далее будем считать, что служба запущена и работает. Чтобы взаимодействовать со службой, необходимо создать сообщение XML в нужном формате, послать сообщение в конечную точку службы, ждать ответного сообщения и, наконец, извлечь данные из документа XML. Благодаря стандарту WSDL и инструментам, таким как `svcutil.exe` и VS 2010, очень легко создать исходный код (прокси), который имитирует поведение службы и обрабатывает все сообщения и детали преобразований в последовательную форму.

Как упоминалось ранее, существуют различные подходы к контролю над выполнением программ. Вам нужно знать, какие части кода должны быть отражены в виде метаданных в WSDL, а не точную спецификацию WSDL. Полезно знать также некоторые основные элементы WSDL, чтобы хорошо понимать, что происходит “за кулисами”, и интерпретировать полученные документы метаданных. Чтобы понять основные идеи взаимодействия элементов в документе WSDL и вашего кода, начните с простого примера, чтобы увидеть, какие фрагменты кода приводят к какому элементу в документе WSDL.

Прокат автомобилей — пример реализации

Пока в целом были объяснены автоматическая генерация метаданных и использование прокси. Теперь настало время увидеть все эти понятия в действии.

Чтобы дать вам краткий и компактный обзор того, как создать соглашение о службе, показана упрощенная реализация вымышленной службы проката автомобилей с подробными пояснениями кода листингов. Если вы выберете подход на основе описания службы, то, скорее всего, начнете с определения документа WSDL, а не с написания кода.

При использовании основанного на коде подхода нужно начать с определения интерфейса .NET и дополнить контракт службы атрибутом `[ServiceContract]`, а ваши методы — атрибутом `[OperationContract]` из пространства имен `System.ServiceModel`.

Шаг 1. Определение контракта службы

Определения контрактов службы, как показано в листинге 2.1, должны находиться в своем собственном модуле, поскольку есть несколько преимуществ в использовании отдельной библиотеки классов для интерфейса и контрактов службы, таких как управление версиями, отделение абстрактных определений от конкретной реализации и распространение.

Листинг 2.1. Определение контрактов службы


```
using System;
using System.ServiceModel;

namespace Wrox.CarRentalService.Contracts
{
  [ServiceContract()]
  public interface ICarRentalService
  {
   [OperationContract]
   double CalculatePrice(DateTime pickupDate, DateTime returnDate,
   string pickupLocation, string vehiclePreference);
  }
}
```

Как видно из листинга 2.1, пример интерфейса службы достаточно прост и содержит всего один метод, `CalculatePrice`, с четырьмя аргументами и возвращаемым значением типа `double`.

Чтобы определить контракт службы, примените атрибут `[ServiceContract]` для определения интерфейса и атрибут `[OperationContract]` для каждого метода, который следует включить в качестве операции службы в документ WSDL. Если интерфейс содержит метод без атрибута `[OperationContract]`, такой метод не будет включен в созданные метаданные службы. Это также называется *моделью включения*.

Можно также аннотировать класс — конкретную реализацию службы — атрибутом `[ServiceContract]`, а его методы — атрибутом `[OperationContract]`. Тем не менее есть несколько причин, почему делать это не рекомендуется. Одна из них заключается в том, что вы будете смешивать "внешний мир" (соглашение о службе) с "внутренним миром" (код реализации), а это нарушает правило о явных границах. Всегда необходимо отделять абстрактное определение службы от конкретной реализации, — только в этом случае можно создать производительный код.

Дополнительно можно создать различные реализации того же соглашения о службе и развернуть их в различных конечных точках службы, если применить данный атрибут на уровне интерфейса.

После компиляции исходного кода получится собранный модуль `Wrox.CarRentalService.Contracts.dll`, который содержит определения соглашений об интерфейсе и службе.

Шаг 2. Извлечение метаданных службы

Чтобы лучше понимать, как каждая часть определения интерфейса соотносится с документом WSDL, в первую очередь создается документ WSDL.

Если служба предоставляет Metadata Exchange Endpoint (MEX), можно сразу перейти к этой конечной точке и посмотреть порожденные метаданные.

Если у вас нет MEX, можно с помощью `svcutil.exe` сгенерировать файл WSDL. Инструмент ServiceModel Metadata Utility Tool (`svcutil.exe`), находящийся в папке установки Windows SDK, предоставляет ряд функциональных возможностей.

- Генерация кода из запущенных служб или статических документов метаданных.
- Экспорт метаданных из скомпилированного кода.
- Проверка скомпилированного кода службы.

- Загрузка метаданных документа из запущенной службы.
- Генерация кода сериализации.

Используйте команду `svcutil.exe Wrox.CarRentalService.Contracts.dll` в командной строке Visual Studio, чтобы получить три файла, приведенных в табл. 2.1.

Таблица 2.1. Файлы WSDL и XSD

Имя файла	Описание
`tempuri.org.wsdl` (см. листинг 2.2)	Документ WSDL со ссылкой на файл XSD, определения сообщений, типы портов служебных операций
`tempuri.org.xsd` (см. листинг 2.3)	Схема XSD для атрибутов метода
`schemas.microsoft.com.2003.10.Serialization.xsd` (см. листинг 2.4)	Стандартная схема XSD для простых типов .NET

Дополнительная информация о схемах содержится на сайте W3C по адресу `http://www.w3.org/XML/Schema`.

Листинг 2.2. WSDL

Доступно для загрузки на Wrox.com

```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/
oasis-200401-wss-wssecurity-utility-1.0.xsd"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
xmlns:tns="http://tempuri.org/"
xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
xmlns:wsap="http://schemas.xmlsoap.org/ws/2004/08/addressing/policy"
xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
xmlns:msc="http://schemas.microsoft.com/ws/2005/12/wsdl/contract"
xmlns:wsa10="http://www.w3.org/2005/08/addressing"
xmlns:wsx="http://schemas.xmlsoap.org/ws/2004/09/mex"
xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
targetNamespace="http://tempuri.org/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <wsdl:types>
    <xsd:schema targetNamespace="http://tempuri.org/Imports">
      <xsd:import namespace="http://tempuri.org/" />
      <xsd:import namespace="http://schemas.microsoft.com/2003/10/
Serialization/" />
  </xsd:schema>
  </wsdl:types>
  <wsdl:message name="ICarRentalService_CalculatePrice_InputMessage">
    <wsdl:part name="parameters" element="tns:CalculatePrice" />
  </wsdl:message>
```

```
  <wsdl:message name="ICarRentalService_CalculatePrice_OutputMessage">
    <wsdl:part name="parameters" element="tns:CalculatePriceResponse" />
  </wsdl:message>
  <wsdl:portType name="ICarRentalService">
    <wsdl:operation name="CalculatePrice">
      <wsdl:input wsaw:Action="http://tempuri.org/ICarRentalService/CalculatePrice"
message="tns:ICarRentalService_CalculatePrice_InputMessage" />
      <wsdl:output wsaw:Action="http://tempuri.org/ICarRentalService/
CalculatePriceResponse" message=
"tns:ICarRentalService_CalculatePrice_OutputMessage" />
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="DefaultBinding_ICarRentalService"
type="tns:ICarRentalService">
    <soap:binding transport="http://schemas.xmlsoap.org/soa
    <wsdl:operation name="CalculatePrice">
      <soap:operation soapAction="http://tempuri.org/ICarRe
CalculatePrice" style="document" />
      <wsdl:input>
        <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  </wsdl:definitions>
```

Листинг 2.3. XSD-файл

Доступно для загрузки на Wrox.com

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:tns="http://tempuri.org/" elementFormDefault="qualified"
targetNamespace="http://tempuri.org/" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="CalculatePrice">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" name="pickupDate" type="xs:dateTime" />
        <xs:element minOccurs="0" name="returnDate" type="xs:dateTime" />
        <xs:element minOccurs="0" name="pickupLocation" nillable="true"
type="xs:string" />
        <xs:element minOccurs="0" name="vehiclePreference"
nillable="true" type="xs:string" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="CalculatePriceResponse">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" name="CalculatePriceResult" type="xs:double" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Листинг 2.4. `schemas.microsoft.com.2003.10.Serialization`


```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:tns=http://schemas.microsoft.com/2003/10/Serialization/
 attributeFormDefault="qualified" elementFormDefault="qualified"
targetNamespace="http://schemas.microsoft.com/2003/10/Serialization/"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="anyType" nillable="true" type="xs:anyType" />
  <xs:element name="anyURI" nillable="true" type="xs:anyURI" />
  <xs:element name="base64Binary" nillable="true"
type="xs:base64Binary" />
  <xs:element name="boolean" nillable="true" type="xs:boolean" />
  <xs:element name="byte" nillable="true" type="xs:byte" />
  <xs:element name="dateTime" nillable="true" type="xs:dateTime" />
  <xs:element name="decimal" nillable="true" type="xs:decimal" />
  <xs:element name="double" nillable="true" type="xs:double" />
  <xs:element name="float" nillable="true" type="xs:float" />
  <xs:element name="int" nillable="true" type="xs:int" />
  <xs:element name="long" nillable="true" type="xs:long" />
  <xs:element name="QName" nillable="true" type="xs:QName" />
  <xs:element name="short" nillable="true" type="xs:short" />
  <xs:element name="string" nillable="true" type="xs:string" />
  <xs:element name="unsignedByte" nillable="true"
type="xs:unsignedByte" />
  <xs:element name="unsignedInt" nillable="true"
type="xs:unsignedInt" />
  <xs:element name="unsignedLong" nillable="true"
type="xs:unsignedLong" />
  <xs:element name="unsignedShort" nillable="true"
type="xs:unsignedShort" />
  <xs:element name="char" nillable="true"
type="tns:char" />
  <xs:simpleType name="char">
    <xs:restriction base="xs:int" />
  </xs:simpleType>
  <xs:element name="duration" nillable="true"
type="tns:duration" />
  <xs:simpleType name="duration">
    <xs:restriction base="xs:duration">
      <xs:pattern value=
"\-?P(\d*D)?(T(\d*H)?(\d*M)?(\d*(\.\d*)?S)?)?" />
      <xs:minInclusive value="-P10675199DT2H48M5.4775808S" />
      <xs:maxInclusive value="P10675199DT2H48M5.4775807S" />
    </xs:restriction>
  </xs:simpleType>
  <xs:element name="guid" nillable="true" type="tns:guid" />
  <xs:simpleType name="guid">
    <xs:restriction base="xs:string">
      <xs:pattern value=
"[\da-fA-F]{8}-[\da-fA-F]{4}-[\da-fA-F]{4}-
[\da-fA-F]{4}-[\da-fA-F]{12}" />
    </xs:restriction>
  </xs:simpleType>
  <xs:attribute name="FactoryType" type="xs:QName" />
```

```
  <xs:attribute name="Id" type="xs:ID" />
  <xs:attribute name="Ref" type="xs:IDREF" />
</xs:schema>
```

Как видно из листингов 2.2 (WSDL) и 2.3 (XSD), различные части XSD и WSDL создаются непосредственно из исходного кода. Чтобы изменить методику присваивания имен по умолчанию и получить дополнительные возможности управления, можно использовать члены атрибутов `[ServiceContract]` и `[OperationContract]`.

В табл. 2.2 показано соответствие между кодом и документами WSDL.

Таблица 2.2. Соответствие между исходным кодом и элементами WSDL

Элемент WSDL	Исходный код
`Message name: ICarRentalService_CalculatePrice`	Имя интерфейса + имя метода + ввод / вывод
`portType name: ICarRentalService`	Имя интерфейса
`Operation name: CalculatePrice`	Имя метода

До сих пор мы создавали метаданные из кода путем определения интерфейса, добавления некоторых методов к нему и дополнения имени интерфейса атрибутом `[ServiceContract]` и методов пользователя — атрибутом `[OperationContract]`. Инфраструктура WCF использует отражение этих данных в созданном документе WSDL.

Шаг 3. Реализация службы

Для завершения служебной части нужна одна или несколько конкретных реализаций интерфейса службы. Конкретные реализации службы рекомендуется помещать в отдельные библиотеки классов. Как показано в листинге 2.5, в реализации кода службы нет необходимости использовать специфические элементы WCF.

Использование атрибутов `[ServiceBehavior]` и `[OperationBehavior]` можно увидеть в коде реализации. Поведение в целом представляет собой внутренние детали реализации и, следовательно, не включается в файл WSDL.

Листинг 2.5. Реализация службы


```
using System;
using Wrox.CarRentalService.Contracts;

namespace Wrox.CarRentalService.Implementations.Europe
{
    public class CarRentalService: ICarRentalService
    {
        public double CalculatePrice
                      (
                      DateTime pickupDate,
                      DateTime returnDate,
                      string pickupLocation,
                      string vehiclePreference
                      )
        {
            // вызов внутренней бизнес-логики
            Random r = new Random(DateTime.Now.Millisecond);
```

```
            return r.NextDouble()*500;
        }
    }
}
```

Шаг 4. Клиент

Метаданные можно использовать для создания необходимого кода клиента. Для этого клиент использует `svcutil.exe` или Add Service Reference в оболочке Visual Studio для получения прокси, который имитирует функционирование службы. (Подробно об этом — в главе 5.) Листинг 2.6 демонстрирует использование созданного прокси в клиентском приложении; листинг 2.7 содержит сообщение SOAP с запросом, а листинг 2.8 — ответное сообщение.

Листинг 2.6. Код клиента


```
using System;
using Wrox.CarRentalService.ConsoleClient.CarRentalProxy;

namespace Wrox.CarRentalService.ConsoleClient
{
    class Program
    {
        static void Main(string[] args)
        {
            using (CarRentalServiceClient
                   carRentalClient = new CarRentalServiceClient())
            {
                double price = carRentalClient.CalculatePrice
                       (
                        DateTime.Now, DateTime.Now.AddDays(5),
                        "Graz", "Pickup"
                       );
                Console.WriteLine("Price {0}",price );
            }
        }
    }
}
```

Листинг 2.7. Сообщение SOAP


```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
 <s:Body>
  <CalculatePrice xmlns="http://tempuri.org/">
   <pickupDate>  2010-01-09T19:15:25.2392813+01:00  </pickupDate>
   <returnDate>  2010-01-14T19:15:25.2402578+01:00  </returnDate>
   <pickupLocation>  Graz  </pickupLocation>
   <vehiclePreference>  Pickup  </vehiclePreference>
  </CalculatePrice>
 </s:Body>
</s:Envelope>
```

Листинг 2.8. Ответное сообщение SOAP


```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
 <s:Body>
  <CalculatePriceResponse xmlns="http://tempuri.org/">
   <CalculatePriceResult>  357.7265724808567  </CalculatePriceResult>
  </CalculatePriceResponse>
 </s:Body>
</s:Envelope>
```

[ServiceContract] и [OperationContract]

Автоматическая генерация документа WSDL иногда требует дальнейшего усовершенствования, — для службы, чтобы предотвратить конфликты имен с другими службами, нужно хотя бы обеспечить осмысленное пространство имен. Иногда применяются некоторые внутренние стили кодирования и шаблоны именования для интерфейсов, классов и методов, которые не разрешены или не рекомендованы в описании службы, или если вы не хотите, чтобы внутренние имена были видимы из внешнего мира.

В целях дальнейшего расширения возможностей управления полученными документами WSDL в табл. 2.3 (из MSDN) приведено подробное описание членов `[ServiceContract]` и `[OperationContract]`.

Таблица 2.3. Члены `[ServiceContract]`

Свойство	Описание
`CallbackContract`	Получает или задает тип обратного вызова, когда соглашение является дуплексной моделью
`ConfigurationName`	Получает или задает имя, используемое для поиска службы в файле конфигурации приложения
`HasProtectionLevel`	Получает значение, указывающее, назначен ли уровень защиты для членов класса
`Name`	Получает или задает имя элемента `<portType>` в WSDL
`Namespace`	Получает или задает пространство имен элемента `<portType>` в WSDL
`ProtectionLevel`	Указывает, обязательно ли соединение для соглашения должно поддерживать значение свойства `ProtectionLevel`
`SessionMode`	Получает или задает, разрешены ли сессии, не разрешены, или требуется

То же самое применяется к атрибуту `[OperationContract]`, как и к атрибуту `[ServiceContract]`, — иногда нужны дополнительные средства управления процессом генерации документа WSDL.

См. таблицу 2.4.

Таблица 2.4. Члены `[OperationContract]`

Параметр	Описание
`Action`	Определяет действие, которое однозначно идентифицирует эту операцию. WCF отправляет сообщения с запросами методам на основе их действий
`AsyncPattern`	Указывает, что операция осуществляется или может быть вызвана асинхронно с помощью пары `Begin/End`

Окончание табл. 2.4

Параметр	Описание
`HasProtectionLevel`	Указывает, что значение `ProtectionLevel` было задано явно
`IsOneWay`	Указывает, что операция состоит только из одного входного сообщения. Операция не генерирует исходящее сообщение
`IsInitiating`	Определяет, может ли эта операция быть начальной в сессии
`ProtectionLevel`	Определяет уровень безопасности, необходимый для выполнения операции
`IsTerminating`	Определяет, будет ли WCF пытаться прекратить текущую сессию после завершения операции

В листинге 2.9 показано, как использовать некоторые общие атрибуты. Другие атрибуты описаны в главе 3.

Листинг 2.9. Атрибуты `[ServiceContract]` и `[OperationContract]`


```
using System;
using System.ServiceModel;

namespace Wrox.CarRentalService.Contracts
{
    [ServiceContract(Namespace="http://wrox/CarRentalService/2009/10",
                     Name="RentalService")]
    public interface ICarRentalService
    {
        [OperationContract(Name = "GetPrice")]
        double CalculatePrice(DateTime pickupDate, DateTime returnDate,
        string pickupLocation, string vehiclePreference);

        [OperationContract( Name = "GetPriceOverloaded")]
        double CalculatePrice(string pickupLocation,
        string vehiclePreference);

        [OperationContract(IsOneWay=true,
        ProtectionLevel=System.Net.Security.ProtectionLevel.None )]
        void UpdatePrice(string vehicleId, double newPrice);
    }
}
```

Результаты использования этих атрибутов отражены во фрагменте WSDL, приведенном в листинге 2.10.

Листинг 2.10. Документ WSDL


```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions . . ..
xmlns:xsd="http://www.w3.org/2001/XMLSchema" targetNamespace=
"http://wrox/CarRentalService/2009/10"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

  <wsdl:message name="RentalService_GetPrice_InputMessage">
    <wsdl:part name="parameters" element="tns:GetPrice" />
  </wsdl:message>
```

```
  <wsdl:message name="RentalService_GetPrice_OutputMessage">
    <wsdl:part name="parameters" element="tns:GetPriceResponse" />
  </wsdl:message>
  <wsdl:message name="RentalService_GetPriceOverloaded_InputMessage">
    <wsdl:part name="parameters" element="tns:GetPriceOverloaded" />
  </wsdl:message>
  <wsdl:message name="RentalService_GetPriceOverloaded_OutputMessage">
  <wsdl:part name="parameters" element="tns:GetPriceOverloadedResponse" />
  </wsdl:message>
  <wsdl:message name="RentalService_UpdatePrice_InputMessage">
    <wsdl:part name="parameters" element="tns:UpdatePrice" />
  </wsdl:message>
  <wsdl:portType name="RentalService">
    <wsdl:operation name="GetPrice">
      <wsdl:input wsaw:Action=
"http://wrox/CarRentalService/2009/10/RentalService/GetPrice"
 message="tns:RentalService_GetPrice_InputMessage" />
      <wsdl:output wsaw:Action=
"http://wrox/CarRentalService/2009/10/RentalService/GetPriceResponse"
 message="tns:RentalService_GetPrice_OutputMessage" />
    </wsdl:operation>
    <wsdl:operation name="GetPriceOverloaded">
      <wsdl:input wsaw:Action=
"http://wrox/CarRentalService/2009/10/RentalService/GetPriceOverloaded"
 message="tns:RentalService_GetPriceOverloaded_InputMessage" />
      <wsdl:output wsaw:Action=
"http://wrox/CarRentalService/2009/10/RentalService
/GetPriceOverloadedResponse"
message="tns:RentalService_GetPriceOverloaded_OutputMessage" />
    </wsdl:operation>
    <wsdl:operation name="UpdatePrice">
      <wsdl:input wsaw:Action=
"http://wrox/CarRentalService/2009/10/RentalService/UpdatePrice"
 message="tns:RentalService_UpdatePrice_InputMessage" />
    </wsdl:operation>
  </wsdl:portType>
</wsdl:definitions>
```

Соглашения о данных

Предыдущие примеры продемонстрировали использование атрибутов `[OperationContract]` и `[ServiceContract]`. Атрибут `[OperationContract]`, по существу, определяет, какие операции предоставляются провайдером службы.

Передаваемые данные зависят от индивидуальных параметров передачи и возвращаемого типа данных. Конкретное значение параметра, как правило, передается от объекта .NET в памяти.

Этот объект преобразуется в соответствующую форму и вкладывается в сообщение SOAP. С другой стороны, параметры извлекаются из сообщения SOAP и предоставляются в виде объекта .NET. Это преобразование осуществляется специальными классами сериализации.

WCF поддерживает различные типы сериализации и десериализации, причем некоторые преобразователи были специально разработаны для WCF, хотя некоторые из них существовали в рамках .NET 1.0 и .NET 2.0 и также могут быть использованы с WCF.

WCF использует класс `DataContractSerializer` из пространства имен `System.Runtime.Serialization` в стандартной комплектации для сериализации или десериализации своих объектов. Этот сериализатор был специально разработан для WCF и является особенно быстрым, эффективным и мощным, что определяет его как предпочтительный выбор. Однако, если ваши требования превышают возможности `DataContractSerializer`, можно вернуться к классической сериализации XML.

`DataContractSerializer` поддерживает различные типы данных, включая приведенные ниже.

- Примитивные типы данных.
- Типы данных с атрибутами `[DataContract]`.
- Классы, которые отмечены как сериализуемые.
- Классы, реализующие интерфейс `IXmlSerializable` из пространства имен `System.Xml.Serialization`.
- Перечисления, коллекции и универсальные коллекции.

В предыдущих примерах не было необходимости в дополнительных уровнях сериализации, потому что примитивные типы данных уже сериализуемы. Однако, если приходится работать со сложными типами данных, такими как параметры методов или возвращаемые значения, нужно использовать соответствующие атрибуты (`[DataContract]` и `[DataMember]`), чтобы определить, как эти данные преобразуются в последовательную форму.

Эти два атрибута из пространства имен `System.Runtime.Serialization`, которые конструируются во время разработки, объясняют классу `DataContractSerializer`, как сериализировать специальные типы. Поскольку структуры передаваемых данных также принадлежат соглашению, конечно, применение атрибутов `[DataContract]` и `[DataMember]` непосредственно влияет на подготовленный документ WSDL. Определения данных включены в соглашение (WSDL) в формате, который не зависит от платформы и языков программирования, а именно XSD. Таким образом, клиент и сервер не могут согласиться на обмен типами данных .NET, но совместимы с учетом нейтральной схемы XML определения формата.

С этой целью в листинге 2.11 приведен простой пример, который не включают использование атрибутов `[DataContract]` и `[DataMember]`, а создает документ WSDL, показанный в листинге 2.12.

Листинг 2.11. Операции службы с простыми типами


```
using System;
using System.ServiceModel;

namespace Wrox.CarRentalService.Contracts
{
    [ServiceContract(
             Namespace = "http://wrox/CarRentalService/2009/10",
             Name = "RentalService")]

    public interface ICarRentalService
    {
        [OperationContract]
        double CheckAvgPricePerDay(string carType);
    }
}
```

Листинг 2.12. Документ WSDL с простыми типами


```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:tns="http://wrox/CarRentalService/2009/10" elementFormDefault=
"qualified"
targetNamespace="http://wrox/CarRentalService/2009/10"
 xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="CheckAvgPricePerDay">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" name="carType"
nillable="true" type="xs:string" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="CheckAvgPricePerDayResponse">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0"
name="CheckAvgPricePerDayResult"
type="xs:double" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

В примере, приведенном в листинге 2.11, ни атрибут `[Serializable]`, ни атрибут `[DataContract]` не использовался с соответствующим атрибутом `[DataMember]`. Это возможно потому, что класс `DataContractSerializer` может работать с простыми .NET типами данных без каких-либо мер предосторожности.

Однако в большинстве случаев в качестве содержимого методов в ориентированной на сообщения архитектуре используются не простые типы данных, вместо этого внимание должно быть уделено структуре передаваемых данных и представления их в формате соглашения.

В этом случае можно использовать атрибуты `[DataContract]` и `[DataMember]`, чтобы сообщить классу `DataContractSerializer`, как должны быть преобразованы его объекты и в какую форму.

Если в операциях используются сложные типы данных без указания классу `DataContractSerializer`, как они должны быть обработаны, все общедоступные члены класса сериализируются автоматически. Это позволяет провести сериализацию POCO (простых объектов CLR) без дополнительных изменений в коде. Вы можете исключить отдельные члены с помощью атрибута `[IgnoreDataMember]`. Листинг 2.13 показывает соглашение о службе, в котором метод называется `CalculatePrice`, входной параметр имеет тип `PriceCalculationRequest`, а тип возвращаемого значения будет `PriceCalculationResponse`.

Листинг 2.13. Контракт службы со сложными типами имен


```
namespace Wrox.CarRentalService.Contracts
{
    [ServiceContract(
              Namespace = "http://wrox/CarRentalService/2009/10",
              Name = "RentalService")]
    public interface ICarRentalService
```

```
    {
        [OperationContract]
        PriceCalculationResponse CalculatePrice
        (
            PriceCalculationRequest request);
        }
    }
}
```

Листинги 2.14 и 2.15 демонстрируют использование POCO, атрибута `[IgnoreDataMember]` и результирующий документ WSDL.

Листинг 2.14. Объекты POCO и `[IgnoreDataMember]`


```
using System;
using System.Runtime.Serialization;
using System.Xml.Serialization;

namespace Wrox.CarRentalService.Contracts
{
    public class PriceCalculationRequest
    {
        public DateTime PickupDate { get; set; }
        public DateTime ReturnDate { get; set; }
        public string PickupLocation { get; set; }
        public string ReturnLocation { get; set; }
        private string VehicleType { get; set; }
        [IgnoreDataMember]
        public string Color { get; set; }
    }
}
```

Листинг 2.15. Документ WSDL


```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:tns="http://schemas.datacontract.org/2004/07/
Wrox.CarRentalService.Contracts"
elementFormDefault="qualified"
targetNamespace="http://schemas.datacontract.org/2004/07/
Wrox.CarRentalService.Contracts"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="PriceCalculationRequest">
    <xs:sequence>
      <xs:element minOccurs="0" name="PickupDate" type="xs:dateTime" />
      <xs:element minOccurs="0" name="PickupLocation"
nillable="true" type="xs:string" />
      <xs:element minOccurs="0" name="ReturnDate" type="xs:dateTime" />
      <xs:element minOccurs="0" name="ReturnLocation"
nillable="true" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
  <xs:element name="PriceCalculationRequest"
nillable="true" type="tns:PriceCalculationRequest" />
  <xs:complexType name="PriceCalculationResponse">
    <xs:sequence>
      <xs:element minOccurs="0" name="Flag" type="xs:int" />
```

```
      <xs:element minOccurs="0" name="Price" nillable="true" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
  <xs:element name="PriceCalculationResponse"
nillable="true" type="tns:PriceCalculationResponse" />
</xs:schema>
```

Однако в очень редких случаях можно сделать общедоступными свои бизнес-объекты, включая все общедоступные свойства.

Важно определить соглашения явно в сценарии SOA и обеспечить контроль над именами, в том числе пространством имен ваших элементов, порядком поступления отдельных атрибутов, и над тем, насколько это необходимо.

Следовательно, соглашение о данных и соглашение о службе вместе служат для формирования формального соглашения между службой и клиентом и предоставляют подробное описание структуры данных, которыми обмениваются клиент и служба.

`DataContractSerializer` использует атрибуты `[DataContract]` и `[DataMember]` для управления точным процессом сериализации. `[DataContract]` является моделью подключения, что означает, что сериализируются только свойства или переменные-члены, которым был явно назначен атрибут `[DataMember]`.

Видимость свойств не имеет никакого значения. В листинге 2.16 демонстрируется использование атрибутов `[DataContract]` и `[DataMember]`. В листинге 2.17 приведена получающаяся часть XSD.

Листинг 2.16. Атрибут [DataContract] и [DataMember]


```
using System;
using System.Runtime.Serialization;
using System.Xml.Serialization;

namespace Wrox.CarRentalService.Contract
{
    [DataContract]
    public class PriceCalculationRequest
    {
        [DataMember]
        public DateTime PickupDate { get; set; }
        [DataMember]
        public DateTime ReturnDate { get; set; }
        [DataMember]
        public string PickupLocation { get; set; }
        [DataMember]
        public string ReturnLocation { get; set; }
        public string Color { get; set; }
    }
}
```

Листинг 2.17. Часть XSD


```
<xs:complexType name="PriceCalculationRequest">
    <xs:sequence>
      <xs:element minOccurs="0" name="PickupDate" type="xs:dateTime" />
      <xs:element minOccurs="0" name="PickupLocation"
```

```
nillable="true" type="xs:string" />
      <xs:element minOccurs="0" name="ReturnDate" type="xs:dateTime" />
      <xs:element minOccurs="0" name="ReturnLocation"
nillable="true" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
```

Характеристики схемы, которые автоматически генерируются WCF, включают следующее.

- Пространства имен: `http://schemas.datacontract.org/2004/07/ + CLR` и `http://schemas.datacontract.org/2004/07/Wrox.CarRentalService.Contracts`.
- Атрибуты расположены в алфавитном порядке
- `<xs:element minOccurs="0" name= PickupDate "`
- `<xs:element minOccurs="0" name="PickupLocation "`
- Значение атрибута `MinOccurs` установлено равным 0, что означает, что этот атрибут не является обязательным
- Примитивные типы данных CLR автоматически отображаются в типы данных XSD. `DateTime PickupDate` становится `xs: dateTime`

Соглашения о данных в деталях

Как вы видели из предыдущих примеров, классам данных, как правило, присваивается атрибут `[DataContract]`, а отдельным переменным-членам — атрибут `[DataMember]`.

Листинг 2.18 демонстрирует дополнительные возможности управления порождением схемы с помощью атрибутов `[DataContract]` и `[DataMember]`.

Листинг 2.18. Атрибуты `[DataContract]` и `[DataMember]`


```
[DataContract(
Name="PriceCalculationRequest", Namespace=
"http://schemas.datacontract.org/2004/07/
Wrox.CarRentalService.Contracts")
]
public class PriceReq
{
   [DataMember(Name="PickupDate",Order=1, IsRequired=true )]
   private DateTime FromDate { get; set; }
   [DataMember(Name = "ReturnDate", Order = 3)]
   public DateTime ToDate{ get; set; }
   [DataMember( Order = 2)]
   public string PickupLocation { get; set; }
   [DataMember(Order = 4)]
   public string ReturnLocation { get; set; }
   public string CarType { get; set; }
}
```

Хотя первоначальное определение класса `PriceCalculationRequest` претерпело существенные изменения, проблемы при сериализации экземпляров по сравнению с подготовленной ранее схемой не возникают. Атрибуты `[DataContract]` и `[DataMember]` создают набор данных XML, который согласуется с соглашением (XSD).

Чтобы проиллюстрировать это, обратимся к вновь подготовленной схеме в листинге 2.19, которая совместима с первоначальной схемой.

Листинг 2.19. Эквивалент XSD


```
<xs:complexType name="PriceCalculationRequest">
   <xs:sequence>
      <xs:element minOccurs="0" name="PickupDate" type="xs:dateTime" />
      <xs:element minOccurs="0" name="PickupLocation"
           nillable="true" type="xs:string" />
      <xs:element minOccurs="0" name="ReturnDate" type="xs:dateTime" />
      <xs:element minOccurs="0" name="ReturnLocation"
           nillable="true" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
```

Следующие наблюдения были сделаны при использовании атрибутов `[DataContract]` и `[DataMember]` из пространства имен `System.Runtime.Serialization`.

- Пространство имен по умолчанию может быть переопределено свойством `namespace`.
- Свойство `name` атрибута `[DataContract]` служит для определения имени сложного типа в XSD.
- Модификатор доступа к переменной-члену класса игнорируется.
- `CarType` не является элементом в модели подключения — сериализируются только переменные-члены, которым явно приписан атрибут `[DataMember]`.
- Если свойство `IsRequired = True`, атрибут далее является обязательным. Если этот атрибут не найден в процессе десериализации, возникает ошибка.
- Порядок может быть определен явно с помощью свойства `order`.
- Свойство `name` атрибута `[DataMember]` используется для определения имени элемента XSD.

Свойства атрибутов `[DataContract]` и `[DataMember]` в основном играют важную роль в управлении версиями или в сценарии взаимодействия. Если, например, нужно сохранить установленную по умолчанию схему XML, желательно избежать каких-либо изменений, за исключением разве что лишь незначительных, в классах .NET, можно попытаться создать совместимые с соглашением о данных признаки и свойства, которые упоминались ранее.

Атрибут `KnownTypes`

В объектно-ориентированном программирования часто используются ссылки на унаследованный, а не на базовый класс. Эта концепция полиморфизма используется весьма часто, хотя, если не будут приняты особые меры предосторожности, в WCF при таком подходе возникает ошибка.

Спецификация метода обычно содержит только базовый класс и, следовательно, определение унаследованных классов тоже не включается в документ WSDL. Естественно, это также означает, что унаследованный класс неизвестен для сгенерированного кода прокси и поэтому может не десериализироваться.

Та же проблема возникает, например, при использовании параметра, представляющего собой неуниверсальную коллекцию (класс), как показано в листинге 2.20.

Листинг 2.20. Аргумент ответного сообщения базового класса и `ArrayList`


```
namespace Wrox.CarRentalService.Contracts
{

    [ServiceContract(
       Namespace = "http://wrox/CarRentalService/2009/10",
       Name = "RentalService")]
    [ServiceKnownType(typeof(PriceCalculationResponseDetailed))]
    public interface ICarRentalService
    {
        [OperationContract]
        PriceCalculationResponse CalculatePrice
        (
            PriceCalculationRequest request
        );

        [OperationContract]
        System.Collections.ArrayList GetPrices();
    }
```

Однако конкретная реализация метода в листинге 2.20 не использует базовый класс `PriceCalculationResponse` в качестве возвращаемого значения, она использует унаследованную версию, которая называется `PriceCalculationResponseDetailed`. В листинге 2.21 показан унаследованный класс и конкретная реализация службы.

Листинг 2.21. Унаследованный класс и реализация службы


```
[DataContract]
public class PriceCalculationResponseDetailed
                : PriceCalculationResponse
{
  [DataMember]
  public string  Currency { get; set; }
}


public PriceCalculationResponse
        CalculatePrice(PriceCalculationRequest  request)
{
   PriceCalculationResponseDetailed resp = null;
   reps=new PriceCalculationResponseDetailed();
   resp.Price = 120;
   resp.Currency = "euro";
   return resp;
}
```

Теперь, если прокси-клиент вызывает метод `CalculatePrice`, возвращаемое значение не ожидается, и поэтому возникает следующая ошибка:

```
There was an error while trying to serialize parameter http://wrox/
CarRentalService/2009/10:CalculatePriceResult. The InnerException message
was 'Type ' Wrox.CarRentalService.Contracts.PriceCalculationResponse-
Detailed ' with data contract name ' PriceCalculationResponseDetailed:http://
schemas.datacontract.org/2004/07/Wrox.CarRentalService.Contracts ' is not
```

expected. Consider using a DataContractResolver or add any types not known statically to the list of known types — for example, by using the KnownTypeAttribute attribute or by adding them to the list of known types passed to DataContractSerializer.'

(Возникла ошибка при попытке сериализации параметров http://wrox/CarRentalService/2009/10: CalculatePriceResult. Сообщение InnerException было "Тип" Wrox.CarRentalService.Contracts.PriceCalculationResponseDetailed" с названием соглашения данных "PriceCalculationResponseDetailed: http://schemas.datacontract.org/2004/07/Wrox.CarRentalService.Contracts ' не ожидается. Попробуйте использовать DataContractResolver или добавьте любые типы, не объявленные статически в списке известных типов, например, с помощью атрибута KnownTypeAttribute, или добавьте их в список известных типов, передаваемых в DataContractSerializer.)

Как показывает сообщение об ошибке, эта проблема может быть решена с помощью атрибута KnownType. Атрибут KnownType позволяет наследовать тип данных, который будет добавлен в соглашение о данных, тем самым содействуя правильной сериализации и десериализации.

Нужно использовать либо атрибут [KnownType] в классе данных, либо атрибут [ServiceKnownType] в определении соглашения о службе или соглашения об операции (независимо от того, хотите ли вы везде использовать унаследованный класс или базовый или разрешить эти типы только для специальных контрактов службы или служебных операций).

Простое использование атрибута [KnownType] существует в перечислении всех производных классов в соглашении о данных для базового класса.

Листинг 2.22 демонстрирует использование атрибута [KnownType].

Листинг 2.22. Атрибут KnownType в DataContract


```
[DataContract]
[KnownType(typeof(PriceCalculationResponseDetailed))]
public class PriceCalculationResponse
{
   [DataMember]
   public double Price { get; set; }
}
```

Этот атрибут сообщает DataContractSerializer, что в соглашение о данных в дополнение к базовому соглашению надо включить унаследованный класс. Иерархия наследования сохраняется и в XSD, как видно из листинга 2.23.

Листинг 2.23. XSD и наследование


```
<xs:complexType name="PriceCalculationResponseDetailed">
  <xs:extension base="tns:PriceCalculationResponse">
   <xs:sequence>
    <xs:element minOccurs="0" name="Currency" type="xs:string" />
   </xs:sequence>
```

Если экземпляр наследуемого класса будет встречаться только в некоторых методах, или если унаследованные версии допускаются во всех операциях соглашения о службе, атрибут [ServiceKnownType] можно использовать либо на уровне службы

(листинг 2.24), либо на уровне операций, как показано в листинге 2.25. Использование [ServiceKnownType] на уровне операций означает, что унаследованный класс является допустимым только для этого метода. Использование атрибута на уровне службы означает, что унаследованные версии являются допустимыми для всех методов.

Листинг 2.24. Атрибут [ServiceKnownType] для одного метода


```
[OperationContract]
[ServiceKnownType(typeof(PriceCalculationResponseDetailed))]
PriceCalculationResponse CalculatePrice(PriceCalculationRequest request);
```

Листинг 2.25. Атрибут [ServiceKnownType] на уровне службы


```
[OperationContract]
[ServiceKnownType(typeof(PriceCalculationResponseDetailed))]
PriceCalculationResponse CalculatePrice(PriceCalculationRequest request);
```

Если загрузить эту информацию в файл подкачки в раздел System.Runtime.Serialization, получится более гибкий вариант KnownType, как показано в листинге 2.26.

Листинг 2.26. Раздел DataContractSerializer в файле конфигурации


```
<system.runtime.serialization>
    <dataContractSerializer>
      <declaredTypes>
        <add type="Wrox.CarRentalService.
           Contracts.PriceCalculationResponse,
           Wrox.CarRentalService.Contracts">
          <knownType type="Wrox.CarRentalService.Contracts.
           PriceCalculationResponseDetailed,
           Wrox.CarRentalService.Contracts"/>
        </add>
      </declaredTypes>
    </dataContractSerializer>
  </system.runtime.serialization>
```

Существует некоторая негибкость в указании предыдущим типам (KnownType, ServiceKnownType, config) WCF, какие унаследованные классы поддерживаются. Например, каждый раз, когда получается новый производный класс, приходится добавлять дополнительный атрибут [KnownType] к основному соглашению. Таким образом, WCF предлагает вам другие возможности для нахождения KnownTypes. Можно не указывать фиксированное множество производных классов, а резервировать имя метода, который возвращает список производных классов. Это может быть сделано как с помощью атрибутов [KnownType], так и с помощью атрибутов [ServiceType]. Листинг 2.27 демонстрирует использование атрибута [KnownType] со статическим методом GetTypes.

Листинг 2.27. Динамические известные типы


```
[DataContract]
[KnownType("GetTypes")]
public class PriceCalculationResponse
```

```
{
   static Type[] GetTypes()
   {
     Type[] t = { typeof(PriceCalculationResponseDetailed) };
     return t;
   }
}
```

Управление версиями соглашений о службе и данных

Разработка программного обеспечения в большой степени определяется текущими изменениями в требованиях и, следовательно, в предложенной функциональности и структуре данных в сообщениях. В принципе, любое незначительное изменение в операциях или данных может привести к изменению в документе WSDL, что приводит к новой версии службы. Однако, вообще говоря, ни организационно, ни технически невозможно постоянно обновлять компоненты службы и реагировать на изменения в предлагаемых службах.

Поэтому в этой главе будут продемонстрированы методы и приемы, которые позволят поддерживать обмен совместимыми сообщениями, несмотря на изменения в соглашениях о службе или данных. Эти изменения также рассматриваются как изменения, которые не приводят к отмене соглашения. Хотя соглашения не должны быть идентичны, важно, чтобы они были согласованы. В случае службы это означает, например, что добавление нового метода не будет приводить к несогласованности, так как клиент не знаком с этим новым методом вообще. То же самое относится к соглашению о данных; если, например, добавляется новое дополнительное поле, потому что `DataContractSerializer` просто игнорируют это поле и заполняет его значениями по умолчанию.

Однако в определенных ситуациях не избежать принятия нового соглашения: например, если изменилось имя операции, или если добавляется обязательное поле. Изменения такого рода приводят к изменению контракта, и тогда необходимо создать новую версию. В этом случае лучше всего адаптировать пространство имен к новой версии и разместить службу на новой конечной точке.

Управление версиями соглашения о данных

И WCF, и `DataContractSerializer` очень терпимы к изменениям в структуре соглашения о данных.

Например, легко добавить дополнительные элементы данных или исключить элементы данных, в которых атрибут `IsRequired` установлен равным `false`. Одним из преимуществ этого является то, что нет необходимости беспокоиться о таких мелких деталях, однако, в среде, ориентированной на службу, важно придерживаться соглашений или, если изменения необходимы, объявлять их формально и официально.

Предположим, что клиент вносит следующие изменения в версии 1 соглашения о службе и данных, показанного в листинге 2.28.

Листинг 2.28. Версия 1 соглашения о службе и данных


```
[ServiceContract(
    Namespace = "http://wrox/CarRentalService/2009/10",
    Name = "RentalService")]
```

```
    public interface ICarRentalService
    {
        [OperationContract]
        PriceCalculationResponse CalculatePrice
        (
          PriceCalculationRequest  request, int priority
        );
    }

[DataContract]
public class PriceCalculationRequest
    {
        [DataMember]
        public DateTime PickupDate { get; set; }
        [DataMember]
        public DateTime ReturnDate { get; set; }
        [DataMember]
        public string PickupLocation { get; set; }
        [DataMember]
        public string ReturnLocation { get; set; }

    }
```

Изменения в операциях и соглашениях о данных, показанные в листинге 2.29, восстанавливают согласованность соглашений.

Листинг 2.29. Версия 2 соглашения о службе и данных


```
[ServiceContract(
    Namespace = "http://wrox/CarRentalService/2009/10",
    Name = "RentalService")]
public interface ICarRentalService
{
   [OperationContract]
   PriceCalculationResponse CalculatePrice
   (
     PriceCalculationRequest  request, string someData
   );
}

[DataContract]
public class PriceCalculationRequest
{
   [DataMember]
   public DateTime PickupDate { get; set; }
   [DataMember]
   public DateTime ReturnDate { get; set; }
   [DataMember]
   public string PickupLocation { get; set; }
   [DataMember]
   public string Color { get; set; }
}
```

Из предыдущего примера можно сделать следующие выводы.

- Параметры метода могут быть удалены или добавлены.
- Элементы данных могут быть удалены или добавлены.

- Типы данных могут быть изменены, если при этом сохранена их совместимость.
- Методы могут быть добавлены.

Эти изменения не приведут к каким-либо проблемам технического характера. Однако необходимо, например, определить детали процедуры обработки пропущенных значений. Вполне возможно, что какое-нибудь конкретное значение больше не понадобится и поэтому может быть проигнорировано без каких-либо проблем.

Управление версиями соглашений о данных "туда и обратно"

Проблема, которая возникает снова и снова в связи с версиями соглашений о данных, состоит в добавлении атрибута, десериализация которого в старом соглашении о данных игнорируется.

Давайте более подробно рассмотрим пример, относящийся к запросу цены, — клиент посылает службе запрос о цене и получает ответ в объекте, содержащем признак и цену. Клиент может подтвердить предложенную цену, снова вызвав метод `ConfirmPrice`, как показано в листинге 2.30.

Листинг 2.30. Соглашение о службе `ConfirmPrice`


```
[ServiceContract(
   Namespace = "http://wrox/CarRentalService/2009/10
   Name = "RentalService")]
public interface ICarRentalService
{
[OperationContract]
PriceCalculationResponse CalculatePrice
   (
     PriceCalculationRequest request
   );
[OperationContract]
bool ConfirmPrice(PriceCalculationResponse resp);
}
[DataContract]
public class PriceCalculationResponse
{
  [DataMember]
  public int Flag { get; set; }
  [DataMember]
  public string Price { get; set; }
}
```

Листинг 2.31. Код клиента `ConfirmPrice`


```
using (RentalServiceClient carRentalClient = new RentalServiceClient())
{
   PriceCalculationRequest req = new PriceCalculationRequest();
   req.PickupDate = System.DateTime.Now.AddDays(5);
   req.ReturnDate = System.DateTime.Now.AddDays(7);
   req.PickupLocation = "Graz";
   req.ReturnLocation = "Villach";
   PriceCalculationResponse resp;
```

```
    resp = carRentalClient.CalculatePrice(req);
    Console.WriteLine("Price {0}", resp.Price);
    Console.WriteLine(carRentalClient.ConfirmPrice(resp));
}
```

При необходимости интернационализации нужно в соглашение о данных добавить атрибут `Currency`.

Ваш служба все еще может использовать код клиента, поскольку дополнительное поле не приводит к каким-либо проблемам, если свойство `IsRequired` не применяется. Однако значение атрибута `Currency` теряется на стороне клиента, поскольку клиент не знаком с этим недавно добавленным атрибутом и соответственно не может десериализировать его содержание.

Интерфейс `IExtensibleDataObject`

Решение проблемы кругового обращения заключается в реализации интерфейса `IExtensibleDataObject` и присвоении статуса, необходимого полю `ExtensionDataObject`. Если `DataContractSerializer` обнаруживает неизвестные элементы в документе XML, они записываются в свойство `ExtensionDataObject` во время десериализации. Содержание `ExtensionDataObject` сохраняется и при дальнейшем использовании этого объекта, в результате чего не происходит потери данных при смене различных версий соглашений о данных.

Если вы используете диалоговое окно Add Service Reference (Добавить ссылку на службу), классы данных на стороне клиента автоматически реализовывают этот интерфейс. Если хотите выполнить ответные действия на стороне сервера, необходимо реализовать этот интерфейс вручную в классах данных; код в листинге 2.32 показывает, как это сделать.

Листинг 2.32. Интерфейс `IExtensibleDataObject`

Доступно для загрузки на Wrox.com

```
[DataContract]
public class PriceCalculationResponse :IExtensibleDataObject
{
    public ExtensionDataObject ExtensionData { get; set; }

    [DataMember]
    public int Flag { get; set; }
    [DataMember]
    public double Price { get; set; }
    [DataMember]
    public string Currency { get; set; }
}
```

Вы также можете защитить `ExtensionData` в целом, установив в файле конфигурации свойство `<DataContractSerializer ignoreExtensionDataObject = "True" />`. Необходимо соблюдать особую осторожность при работе с расширением данных на стороне сервера, поскольку это может пробить брешь в безопасности.

Рекомендации по управлению версиями соглашения о службе

Если требуется точное соответствие между схемами, необходимо присвоить новую версию вашему соглашению при каждой новой реализации изменений.

Если нет необходимости точного соответствия между схемами, вы должны учесть следующие моменты.

- В любое время можно добавить новые методы.
- Можно удалять любые существующие методы.
- Типы данных параметров должны оставаться совместимыми.

Рекомендации по управлению версиями соглашения о данных

Если требуется точное согласование схем, каждый раз, когда вы выполняете реализацию изменений, вы должны присвоить вашему соглашению новую версию.

Если нет необходимости в точном согласовании схем, нужно учесть следующие моменты.

- Соглашениям о данных не обязательно присваивать новые версии из-за наследования. Вместо этого создайте новый независимый класс данных.
- Для облегчения кругового обмена данными следует в самом начале реализовать интерфейс `IExtensibleDataObject`.
- Если необходимо изменить имя класса данных или некоторых его членов, с помощью атрибутов `DataContract` и `DataMember` можно создать совместимые соглашения о данных.
- Нельзя вносить последующие изменения в типы данных.
- Нельзя изменять порядок появления членов данных с помощью свойства `[DataMember (Order=?)]`.
- Нельзя изменять присвоенное по умолчанию значение `IsRequired` (`false`).
- В любое время можно добавить дополнительные элементы данных, но это изменит порядок сериализации. Чтобы избежать этой проблемы, можно установить свойство `Order` для новых членов равным значению в текущей версии. Таким образом, данным-членам, добавленным в версии 2, должно быть присвоено значение `Order = 2`.
- Данные-члены не должны удаляться.
- Свойство `IsRequired` не должно подвергаться последующим изменениям.

Листинг 2.33 демонстрирует некоторые рекомендации.

Листинг 2.33. Оптимальные способы управления версиями

Доступно для загрузки на Wrox.com

```
[DataContract(
   Name="PriceCalculationRequest",
   Namespace="http://schemas.datacontract.org/2004/07/
   Wrox.CarRentalService.Contracts")]
public class PriceReq //Version 1
{
   [DataMember()]
   private DateTime PickupDate { get; set; }
   [DataMember()]
   public DateTime ReturnDate { get; set; }
   [DataMember()]
   public string PickupLocation { get; set; }
   [DataMember()]
   public string ReturnLocation { get; set; }
```

```
}

[DataContract(
   Name = "PriceCalculationRequest",
   Namespace = "http://schemas.datacontract.org/2004/07/
   Wrox.CarRentalService.Contracts")]
public class PriceReq // Version 2
   [DataMember(Name = "PickupDate")]
   private DateTime FromDate { get; set; }
   [DataMember(Name = "ReturnDate")]
   public DateTime ToDate { get; set; }
   [DataMember(Order = 2)]
   public string PickupLocation { get; set; }
   [DataMember()]
   public string ReturnLocation { get; set; }
   [DataMember(Order=2)] // из Version 2
   public string CarType { get; set; }
}
```

Соглашения о сообщениях

В начале этой главы было разъяснено, что сообщения SOAP передаются между клиентом и сервером. Однако до сих пор вы могли только влиять на содержимое тела сообщения SOAP. В большинстве же случаев рекомендуется управлять соглашениями о данных и операциях. Но в некоторых ситуациях вы можете получить полный контроль над всем сообщением SOAP (заголовок и тело). Заголовок SOAP является подходящим методом для передачи дополнительных данных без расширения класса данных (например, признак защиты).

Соглашения о сообщениях позволяют полностью управлять содержанием заголовка SOAP, так же как и структурой тела сообщения SOAP. До сих пор соглашения о данных использовались как передаваемые или возвращаемые параметры, но теперь они просто заменяются соглашениями о сообщениях. Если вы предпочтете соглашения о сообщениях, важно использовать соглашения о сообщениях для всех параметров, так как методы не могут содержать смесь соглашений о данных и сообщениях.

Листинг 2.34 показывает, как используются три атрибута: `[MessageContract]`, `[MessageHeader]` и `[MessageBody]`. В дополнение к объекту запроса цены в заголовке SOAP также передается имя пользователя.

Листинг 2.34. Атрибуты `[MessageContract]`, `[MessageHeader]` и `[MessageBody]`

Доступно для загрузки на Wrox.com

```
[ServiceContract(
   Namespace = "http://wrox/CarRentalService/2009/10",
   Name = "RentalService")]
public interface ICarRentalService
{
   [OperationContract]
   PriceCalculationResp  CalculatePrice(PriceCalculation  request);
}

[MessageContract]
public class PriceCalculation
{
```

```
   [MessageHeader]
   public CustomHeader SoapHeader { get; set; }

   [MessageBodyMember]
   public PriceCalculationRequest PriceRequest { get; set; }
}

[DataContract]
public class CustomHeader
{
   [DataMember]
   public string Username { get; set; }
}

[DataContract]
public class PriceCalculationRequest
{
   [DataMember]
   public DateTime PickupDate { get; set; }
   [DataMember]
   public DateTime ReturnDate { get; set; }
   [DataMember]
   public string PickupLocation { get; set; }
   [DataMember]
   public string ReturnLocation { get; set; }
     }
```

В листинге 2.35 приводится использование соглашения о сообщениях на стороне клиента, а в листинге 2.36 приводится сообщение SOAP.

Листинг 2.35. Соглашения о сообщениях на стороне клиента


```
using (RentalServiceClient carRentalClient = new RentalServiceClient())
{
   PriceCalculation calc = new PriceCalculation();
   PriceCalculationRequest req = new PriceCalculationRequest();
   req.PickupDate = System.DateTime.Now.AddDays(5);
   req.ReturnDate = System.DateTime.Now.AddDays(7);
   req.PickupLocation = "Graz";
   req.ReturnLocation = "Villach";
   calc.PriceRequest = req;

   CustomHeader sHeader = new CustomHeader();
   sHeader.Username = "Johann";
   calc.SoapHeader = sHeader;

   PriceCalculationResp resp=  carRentalClient.CalculatePrice(calc);
}
```

Листинг 2.36. SOAP и заголовки сообщений


```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
 <s:Header>
  <h:SoapHeader xmlns:h="http://wrox/CarRentalService/2009/10"
  xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
```

```
    <Username xmlns="http://schemas.datacontract.org/2004/07/
   Wrox.CarRentalService.Contracts">
    Johann
    </Username>
  </h:SoapHeader>
 </s:Header>
<s:Body>
 <PriceCalculation xmlns="http://wrox/CarRentalService/2009/10">
  <PriceRequest
 xmlns:a=
"http://schemas.datacontract.org/2004/07/
Wrox.CarRentalService.Contracts"
 xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
   <a:PickupDate> 2010-01-15T14:15:08.1683905+01:00 </a:PickupDate>
   <a:PickupLocation> Graz </a:PickupLocation>
   <a:ReturnDate> 2010-01-17T14:15:08.1693671+01:00 </a:ReturnDate>
   <a:ReturnLocation> Villach </a:ReturnLocation>
  </PriceRequest>
 </PriceCalculation>
</s:Body>
</s:Envelope>
```

Другие примеры использования соглашения о сообщениях приведены в главе 7.

XML сериализации

Как уже упоминалось во введении, WCF поддерживает несколько типов сериализации.

По умолчанию WCF использует класс `DataContractSerializer` для сериализации ваших объектов CLR. В предыдущих примерах вы видели также использование различных свойств в связи с атрибутами `[DataContract]` и `[DataMember]`. Кроме того, вы можете управлять структурой сгенерированного набора информации XML.

`DataContractSerializer` оптимизирован по производительности, поэтому он предлагает меньше возможностей, поскольку не учитывает влияния на структуру данных. Если эти возможности не соответствуют вашим потребностям, можно вернуться к сериализатору XML. Сериализатор XML предлагает большую свободу в сценариях совместной работы и взаимодействии со старыми веб-службами ASP.NET.

Чтобы указать, что WCF должен использовать сериализатор XML, можно просто присвоить соглашению о службе атрибут `[XmlSerializerFormat]`. Если же нужно использовать сериализатор XML только для некоторых операций, к ним также можно применить этот атрибут. Заметьте, что сериализатор XML также сериализирует все общие свойства по умолчанию. Более подробно эти классы описаны в стандартизованной документации на MSDN.

В листинге 2.37 показано использование `[XMLSerializerFormat]` с атрибутами из пространства имен `System.XML.Serialization`, а в листинге 2.38 приведен файл XSD для типа `PriceCalculationRequest`.

Листинг 2.37. Сериализация XML


```
[XmlSerializerFormat]
[ServiceContract(
   Namespace = "http://wrox/CarRentalService/2009/10",
```

```
   Name = "RentalService")]
public interface ICarRentalService
{
   [OperationContract]
   PriceCalculationResponse CalculatePrice
   (
    PriceCalculationRequest request
   );
}

[XmlRoot("PriceRequest")]
public class PriceCalculationRequest
{
   [XmlAttribute()]
   public string PickupLocation { get; set; }
   [XmlElement(ElementName="ReturnLoc")]
   public string ReturnLocation { get; set; }
   [XmlIgnore()]
   public DateTime PickupDate { get; set; }
   private DateTime ReturnDate { get; set; }
}
```

Листинг 2.38. XSD для типа `PriceCalculationRequest`


```
<xs:element name="CalculatePrice">
 <xs:complexType>
  <xs:sequence>
<xs:element minOccurs="0" maxOccurs="1"
 name="request"
 type="tns:PriceCalculationRequest" />
</xs:sequence>

</xs:complexType>
</xs:element>
<xs:complexType name="PriceCalculationRequest">
 <xs:sequence>
  <xs:element minOccurs="0" maxOccurs="1" name="ReturnLoc"
  type="xs:string" />
  </xs:sequence>
 <xs:attribute name="PickupLocation" type="xs:string" />
</xs:complexType>
<xs:element name="CalculatePriceResponse">
 <xs:complexType>
  <xs:sequence>
   <xs:element minOccurs="0" maxOccurs="1"
 name="CalculatePriceResult"
 type="tns:PriceCalculationResponse" />
  </xs:sequence>
 </xs:complexType>
</xs:element>
.
</xs:complexType>
```