ГЛАВА 36

Стандарты и расширения веб-служб

В предыдущей главе было показано, как .NET скрывает низкоуровневый механизм веб-служб, тем самым позволяя создавать и использовать сложнейшие веб-службы, не зная ничего о низкоуровневых деталях используемых протоколов. Такая высокоуровневая абстракция является одной из основных тенденций в современном программировании: например, Windows-разработчикам редко когда приходится волноваться об отдельных элементах в их бизнес-приложениях, равно как и ASP.NET-разработчикам редко когда приходится волноваться о составлении базовой разметки для выходного потока. Конечно, иногда одних высокоуровневых структур бывает недостаточно. Например, Windows-разработчикам, занимающимся созданием функционирующих в реальном времени видеоигр, пожалуй, просто не избежать взаимодействия с низкоуровневым видеооборудованием, а веб-разработчикам, занимающимся созданием специальных элементов управления, скорее всего, придется "погружаться в дебри" JavaScript и HTML.

Упомянутый принцип работает и в случае веб-служб. Чаще всего высокоуровневой модели веб-служб в .NET будет вполне хватать. Она обеспечивает быстрое, продуктивное и защищенное от ошибок кодирование. Однако в некоторых случаях придется "копать немного глубже", в частности, это может случиться при наличии необходимости в отправке сложных объектов клиентам, отличным от .NET, или создании расширений, которые бы соответствовали .NET-модели веб-служб. В этой главе мы посмотрим, как выглядит этот более низкий уровень, а также более подробно расскажем о лежащих в основе протоколах SOAP и WSDL.

Совет. Тем читателям, кто хорошо разбирается в веб-службах и которых интересуют новые функциональные возможности в .NET 2.0, стоит внимательно прочитать находящийся ближе к концу этой главы раздел "Настройка SOAP-сообщений". В этом разделе рассказывается о том, как можно управлять процессом сериализации с помощью интерфейса `IXmlSerializable` и как создавать расширения импортера схем, чтобы добавить типы, которые веб-службы обычно не поддерживают. Также стоит прочитать раздел "Реализация существующего контракта", в котором показано, как выполняется разработка на основе контрактов с помощью WSDL.

Стандарт WS-Interoperability

Технология веб-служб развивалась достаточно стремительно, а некоторые ее стандарты, такие как SOAP и WSDL, все еще продолжают развиваться. В первых наборах инструментальных средств для веб-служб некоторые части этих стандартов интерпретировались различными производителями по-разному, что привело к возникновению проблем с функциональной совместимостью. Более того, некоторые из функциональных возможностей, которые были доступны в исходной версии этих стандартов, сейчас уже считаются устаревшими.

Поиск компромисса между такими незначительными отличиями — задача не из легких, особенно, когда необходимо создать веб-службу или веб-службы, доступ к которым будут получать клиенты, использующие другие платформы программирования и инструментальные наборы создания веб-служб. К счастью, недавно появился еще один стандарт, который охватывает огромное количество различных правил и рекомендаций и ориентирован на обеспечение функциональной совместимости между реализациями веб-служб различных производителей. Этот документ называется "WS-Interoperability Basic Profile" ("Базовый профиль функциональной совместимости веб-служб"), подробное описание которого можно найти по адресу: `http://www.ws-i.org`. Он содержит рекомендованный набор спецификаций SOAP 1.1 и WSDL 1.1, а также несколько основных правил. Стандарт WS-Interoperability поддерживается всеми производителями веб-служб (включая Microsoft, IBM, Sun и Oracle).

В идеале разработчики не должны волноваться о специфических особенностях стандарта WS-Interoperability. Вместо этого .NET должен учитывать его правила внутренне. В .NET 2.0 этот вопрос решается с помощью атрибута `WebServiceBinding`, который автоматически добавляется в класс веб-службы при его создании в Visual Studio:

```
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
public class MyService : WebService
{ ... }
```

Атрибут `WebServiceBinding` обозначает уровень совместимости, на который рассчитывает разработчик. На текущий момент доступна только одна опция — `WsiProfiles.BasicProfile1_1`, которая представляет стандарт WS-Interoperability Basic Profile 1.1. Однако по мере развития стандартов будут появляться все более новые версии протоколов SOAP и WSDL, а также профиля WS-Interoperability, который идет вместе с ними.

После добавления атрибута `WebServiceBinding` .NET предупредит (путем выдачи соответствующей ошибки во время компиляции), если веб-служба выходит за рамки допустимого поведения. По умолчанию все веб-службы, разработанные с использованием .NET, являются совместимыми, но разработчик, добавив те или иные атрибуты, может случайно создать и несовместимую службу. Например, не исключено появление двух веб-методов с одинаковыми именами, если их сигнатура отличается и им с помощью свойства `MessageName` атрибута `WebMethod` присваиваются разные имена сообщений. Однако согласно стандарту WS-Interoperability такое поведение является недопустимым, поэтому при попытке запустить такую веб-службу .NET сгенерирует страницу ошибки, подобную той, которая показана на рис. 36.1.

Также разработчик может решить прорекламировать совместимость своей веб-службы. Он может сделать это с помощью свойства `EmitConformanceClaims`, как показано ниже:

```
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1,
EmitConformanceClaims=true)]
public class MyService : WebService
{ ... }
```

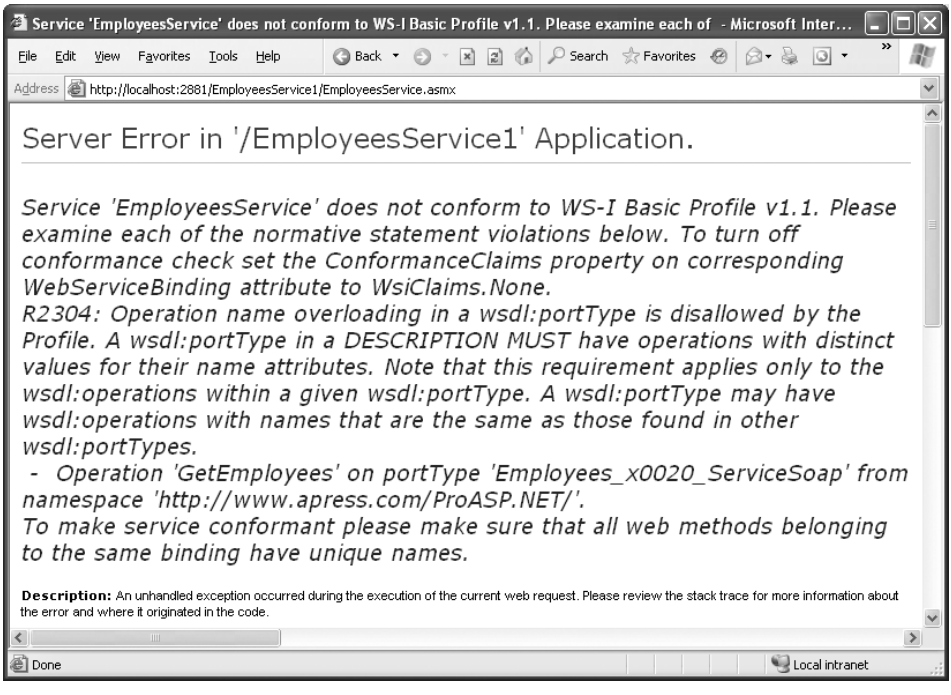

Рис. 36.1. SOAP-сообщение

В таком случае в WSDL-документ помещается дополнительная информация, указывающая на то, что эта веб-служба соответствует стандарту. Важно понимать, что это делается исключительно в информационных целях: веб-служба может отвечать стандарту, даже когда об этом нигде явно не сообщается.

В редких случаях разработчик может принять решение нарушить какое-нибудь из правил стандарта WS-Interoperability, например, чтобы создать веб-службу, которой бы могло пользоваться какое-то более старое, несовместимое с этим стандартом приложение. В такой ситуации ему первым делом потребуется отключить функцию проверки совместимости за счет удаления атрибута `WebServiceBinding`. Или же он может отключить проверку совместимости и документировать этот факт, используя атрибут `WebServiceBinding` без профиля:

```
[WebServiceBinding(ConformsTo = WsiProfiles.None)]
public class MyService : WebService
{ ... }
```

Стандарт SOAP

SOAP — это межплатформенный стандарт, который используется для форматирования сообщений, пересылаемых между веб-службами и клиентскими приложениями. Красота стандарта SOAP заключается в его гибкости. Вы можете не только применять его для отправки XML-данных любого типа (включая ваши собственные оригинальные XML-документы), но также использовать его с транспортными протоколами, отличными от HTTP. Например, вы могли бы пересылать SOAP-сообщения через прямое TCP/IP-соединение.

На заметку! .NET Framework включает поддержку только для наиболее распространенного способа использования SOAP, которым является применение SOAP через HTTP. При желании получить большую гибкость, попробуйте установить предлагаемый Microsoft набор инструментальных средств WSE (Web Services Enhancements), о котором более подробно будет рассказываться в следующей главе.

SOAP — это достаточно простой стандарт. Принцип номер один: каждое SOAP-сообщение представляет собой XML-документ. Этот XML-документ имеет один единственный корневой элемент, которым является SOAP-конверт. Остальные данные для сообщения хранятся внутри конверта, который состоит из раздела заголовка и непосредственно тела сообщения.

На заметку! Первоначально SOAP считалось аббревиатурой (которая расшифровывалась как Simple Object Access Protocol — простой протокол доступа к объектам). Однако в текущих версиях стандарта SOAP такого больше нет. Детальное описание спецификаций стандарта SOAP можно найти по адресу `http://www.w3.org/TR/soap`.

В целом веб-службы .NET используют два типа SOAP-сообщений. Клиент отправляет веб-службе *сообщение запроса* для инициализации веб-метода. По завершении процесса обработки веб-служба отправляет клиенту *ответное сообщение*.

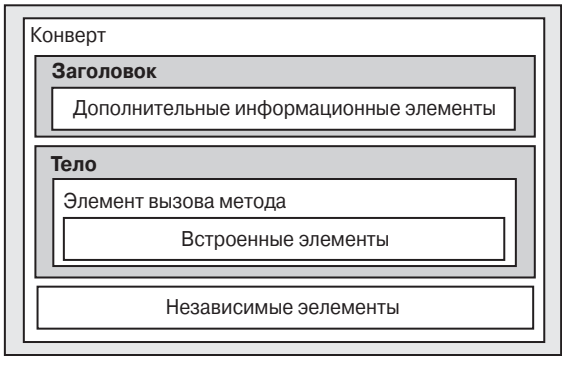

Рис. 36.2. SOAP-сообщение

SOAP-кодировка

Существует два разных (но тесно связанных между собой) типа SOAP. SOAP *документного типа* рассматривает пересылаемые данные как документы. Другими словами, в теле каждого получаемого или отправляемого SOAP-сообщения содержится XML-документ. SOAP *типа RPC* трактует процесс обмена данными как вызовы методов на удаленных объектах. Удаленным объектом может быть Java-объект, COM-компонент, .NET-объект или что-то совершенно другое. При использовании SOAP типа RPC в запросе после имени метода всегда указывается имя самого внешнего элемента, а также значения для каждого параметра этого метода. В ответе имя самого внешнего элемента соответствует имени метода, за которым следует слово `Response`.

Тот факт, что разрабатываемые в .NET веб-службы охватывают объектно-ориентированную модель RPC, может подтолкнуть к выводу о том, что такие веб-службы используют SOAP типа RPC. Однако это не так. Причина достаточно проста: SOAP документного типа является намного более гибким (он допускает пересылку любых XML-документов

между веб-службой и использующим ее приложением). Однако, несмотря на то, что .NET использует стандарт SOAP документного типа, сообщения он форматирует способом, подобным способу стандарта SOAP типа RPC, применяя многие такие же правила.

Еще интереснее стандарт SOAP делает также и тот факт, что данные в SOAP-сообщении могут кодироваться двумя способами: литерально и в соответствии с пятым разделом SOAP-спецификации. Первый тип кодировки означает, что данные кодируются согласно какой-нибудь определенной XML-схемы, а второй — что они кодируются согласно похожим, но более жестким правилам, которые перечислены в пятом разделе SOAP-спецификации. Правила, перечисленные в пятом разделе SOAP-спецификации, являются несколько устаревшими. Причина, по которой они до сих пор существуют, состоит в том, что стандарт SOAP был разработан до завершения разработки стандарта XML Schema.

На заметку! По умолчанию все разрабатываемые в .NET веб-службы используют стандарт SOAP документного типа с литеральной кодировкой. Изменять это поведение следует только при необходимости в совместимости с каким-нибудь унаследованным приложением.

Сейчас вас наверняка интересует вопрос о том, зачем вам нужно знать все эти подробности о стандарте SOAP. В большинстве случаев знать их не обязательно. Однако иногда вам может понадобиться изменить общую кодировку веб-службы. Одной из причин для этого может стать необходимость отобразить веб-метод, который должен вызываться клиентом, поддерживающим только SOAP типа RPC. Хотя такой сценарий становится все более редким явлением (и нарушает правила стандарта WS-Interoperability), все-таки он встречается.

ASP.NET имеет два атрибута (оба они расположены в пространстве имен `System.Web.Services.Protocols`), которые могут использоваться для управления общей кодировкой всех методов в веб-службе.

- **`SoapDocumentService`**. Этот атрибут следует добавлять при желании сделать так, чтобы каждая веб-служба использовала стандарт SOAP документного типа (который, в принципе, уже используется по умолчанию). Однако можно также добавить и параметр `SoapBindingUse`, чтобы указать, что вместо документной кодировки должна применяться кодировка, отвечающая правилам из пятого раздела SOAP-спецификации.
- **`SoapRpcService`**. Этот атрибут следует добавлять при желании сделать так, чтобы каждая веб-служба использовала стандарт SOAP с кодировкой по правилам из пятого раздела SOAP-спецификации.

Для управления кодировкой отдельных методов можно использовать следующие два атрибута.

- **`SoapDocumentMethod`**. Этот атрибут следует добавлять при желании использовать стандарт SOAP документного типа только для какого-то одного конкретного веб-метода. Также можно задать и то, какая кодировка должна при этом применяться.
- **`SoapRpcMethod`**. Этот атрибут следует добавлять при желании использовать стандарт SOAP типа RPC только для какого-то одного конкретного веб-метода.

Это удобно, когда в веб-службе необходимо отобразить два метода, выполняющие схожие функции, но поддерживающие разный тип SOAP-кодировки. В этой главе основное внимание будет уделяться стилю SOAP-сообщений, который .NET использует по умолчанию (т.е. документному с литеральной кодировкой).

Версии SOAP

Наиболее часто используемая версия SOAP на сегодняшний день — это SOAP 1.1. Единственным другим возможным вариантом является появившаяся недавно версия SOAP 1.2, которая проливает свет на многие аспекты стандарта SOAP, содержит некоторые уточнения и формализует модель расширяемости.

На заметку! Какая бы версия SOAP не использовалась, на возможностях веб-служб .NET это никак не отражается. На самом деле, обращать внимание на номер используемой версии следует только в том случае, когда требуется наличие гарантии в совместимости с клиентами, отличными от .NET. В примерах, рассматриваемых в этой главе, применяется версия SOAP 1.1.

Версии .NET 1.*x* поддерживали только SOAP 1.1. Однако веб-служба, создаваемая в .NET 2.0, автоматически поддерживает как SOAP 1.1, так и SOAP 1.2. При желании изменить это поведение, вы можете отключить одну из этих версий с помощью файла `web.config`:

```
configuration>
 <system.web>
 <webServices>
 <protocols>
 <!-- Чтобы отключить SOAP 1.2, используйте такой код -->
 <remove name="HttpSoap12"/>
 <!-- Чтобы отключить SOAP 1.1, используйте такой код -->
 <remove name="HttpSoap"/>
 </protocols>
 </webServices>
 ...
 </system.web>
</configuration>
```

При создании прокси-класса в .NET версия SOAP 1.1 используется по умолчанию, только если версия SOAP 1.2 не доступна. Вы можете переопределить это поведение программно, установив значение для свойства `SoapVersion` прокси-класса, прежде чем вызывать какие-либо веб-методы:

```
proxy.SoapVersion = System.Web.Services.Protocols.SoapProtocolVersion.Soap12;
```

Трассировка SOAP-сообщений

Прежде чем в подробностях рассматривать стандарт SOAP, не помешает узнать, как просматривать SOAP-сообщения, отправляемые веб-службе .NET и из нее. К сожалению, .NET не включает никаких средств для трассировки или отладки SOAP-сообщений. Тем не менее, их достаточно просто просматривать с помощью других средств.

Первый подход — воспользоваться отображаемой в браузере тестовой страницей. Как вы знаете, протокол SOAP эта страница не использует: вместо этого она применяет упрощенный протокол HTTP POST, который кодирует данные в виде пар “имя-значение”. Однако она действительно включает пример того, как должно выглядеть SOAP-сообщение для конкретного веб-метода.

Для примера возьмем веб-службу `EmployeesService`, которую мы разрабатывали в предыдущей главе. Если вы загрузите тестовую страницу, щелкнете на ссылке для метода `GetEmployeesCount()` и прокрутите появившуюся после этого страницу немного вниз, вы увидите пример сообщения запроса и ответа, в которых вместо значений данных будут отображаться указатели места заполнения. Часть этой страницы можно

видеть на рис. 36.3. Прокрутив эту страницу еще немного вниз, вы сможете увидеть формат более простых сообщений HTTP POST.

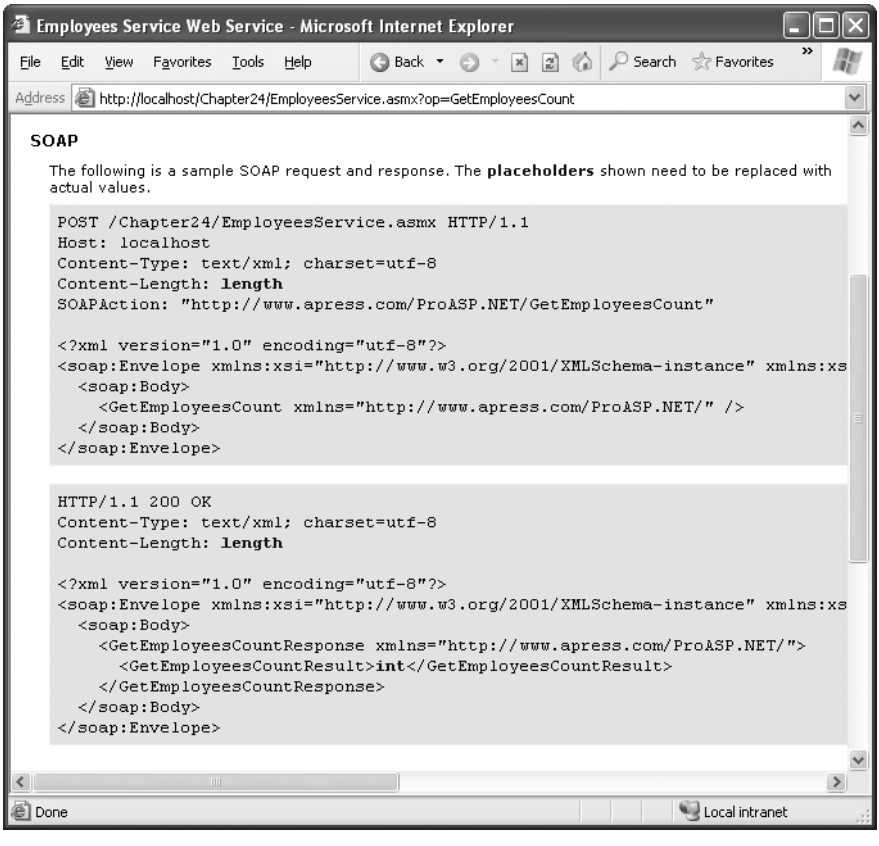

Рис. 36.3. Примеры SOAP-сообщений для метода `GetEmployeesCount()`

Эти примеры помогут разобраться в стандарте SOAP, но от них мало пользы, когда требуется увидеть "реальные" SOAP-сообщения, возможно, чтобы выявить какую-нибудь непредвиденную проблему с совместимостью между веб-службой .NET и клиентом. К счастью, существует простой способ перехватывать настоящие SOAP-сообщения в процессе их передачи, но для этого придется использовать другое средство. Этим средством является Microsoft SOAP Toolkit — COM-библиотека, которая содержит объекты, позволяющие использовать веб-службы в основанных на COM языках, таких как Visual Basic 6 и Visual C++ 6. Кроме этих объектов SOAP Toolkit также включает очень важный инструмент трассировки, позволяющий заглядывать внутрь процесса SOAP-коммуникации.

Загрузить набор инструментальных средств SOAP Toolkit можно по адресу `http://msdn.microsoft.com/webservices/_building/soaptk`. Установив набор SOAP Toolkit, вы можете запустить утилиту трассировки, выбрав в меню Start (Пуск) команду Microsoft SOAP Toolkit⇨Trace Utility (Microsoft SOAP Toolkit⇨Утилита трассировки). Как только утилита трассировки загрузится, выберите в меню File (Файл) команду New⇨Formatted Trace (Создать⇨Форматируемая трассировка). Появится окно, показанное на рис. 36.4.

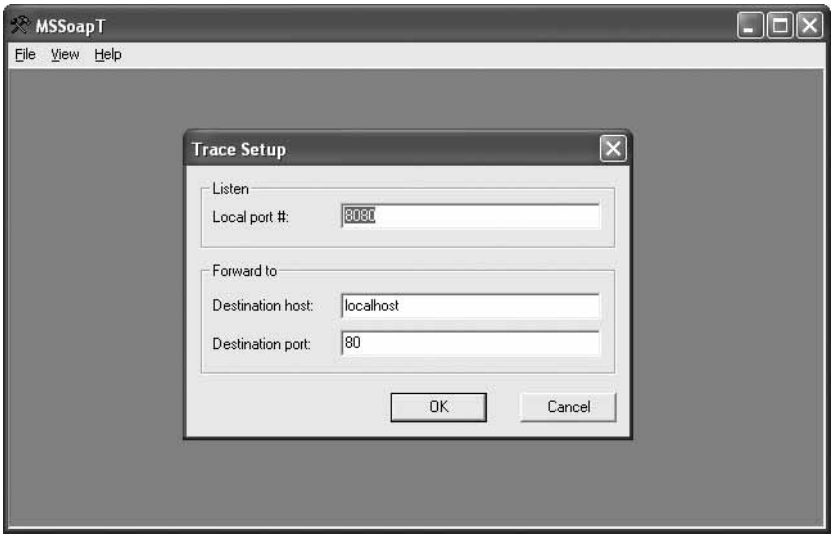

Рис. 36.4. Создание новой SOAP-трассировки

Настройки по умолчанию обозначают, что утилита трассировки будет прослушивать порт 8080 и пересылать все сообщения на порт 80 (который прослушивается веб-сервером IIS для выявления незашифрованного HTTP-трафика, запросов GET и POST и SOAP-сообщений включительно). Щелкните на кнопке OK, чтобы принять эти настройки.

Помимо прочего, требуется еще одна дополнительная деталь. По умолчанию клиенты веб-службы будут обходить утилиту трассировки, посылая свои SOAP-сообщения напрямую в порт 80, а не 8080. Вам потребуется настроить клиентский код так, чтобы он пересылал сообщения через порт 8080, а не 80. Чтобы сделать это, необходимо просто изменить URL-адрес, в котором указывается порт, как показано ниже:

```
http://localhost:8080/MyWebSite/MyWebService.asmx
```

Чтобы изменить URL-адрес, вы должны изменить значение свойства `Url` прокси-класса, прежде чем вызывать какие-либо из его методов. Вместо того чтобы жестко кодировать новый URL-адрес, вы можете воспользоваться показанным ниже кодом, который с помощью класса `System.Uri` в общем переадресует любой URL-адрес на порт 8080:

```
// Создаем прокси.
EmployeesService proxy = new EmployeesService();
Uri newUrl = new Uri(proxy.Url);
proxy.Url = newUrl.Scheme + "://" + newUrl.Host + ":8080" + newUrl.AbsolutePath;
// Вызываем веб-службу и извлекаем результаты.
DataSet ds = proxy.GetEmployeesCount();
```

В код веб-службы такое изменение вносить не нужно, потому что она автоматически пересылает свои ответные сообщение в тот же порт, куда поступают сообщения запроса (в данном случае — в порт 8080), где они регистрируются утилитой трассировки, после чего пересылаются дальше клиентскому приложению.

Закончив вызывать веб-службу, вы можете развернуть дерево в окне утилиты трассировки и просмотреть сообщения запроса и ответа. На рис. 36.5 показаны результаты, получаемые при выполнении приведенного выше фрагмента кода. В верхней части окна отображается сообщение запроса для метода `GetEmployeesCount()`, а в нижней — ответ на него, в котором сообщается, что на текущий момент в таблице содержатся дан-

ные о девяти сотрудниках. По мере вызова веб-методов количество узлов в дереве будет увеличиваться.

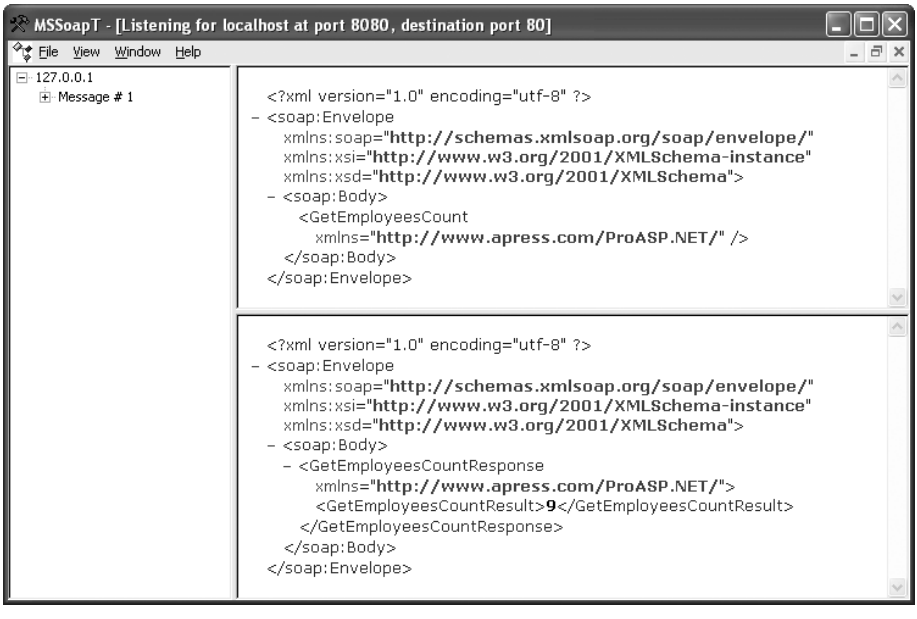

Рис. 36.5. Перехват SOAP-сообщений

Доступная в SOAP Toolkit утилита трассировки является мощным средством для просмотра SOAP-сообщений, особенно, когда необходимо увидеть, как обрабатываются необычные типы данных, протестировать какое-то специальное расширение или локализовать проблему с функциональной совместимостью. Лучше всего то, что вам не нужно устанавливать никакого специального программного обеспечения на веб-сервере. Вместо этого вам просто следует пересылать сообщения клиента в локальную утилиту трассировки.

В следующих разделах вы более подробно ознакомитесь с тем, что собой представляет формат SOAP-сообщений. Утилита трассировки может понадобиться для тестирования приведенных примеров, дабы посмотреть, как выглядят лежащие в их основе SOAP-сообщения.

SOAP-конверт

Каждое SOAP-сообщение помещается в корневой элемент `<Envelope>`. Внутри этого элемента (конверта) находится обязательный элемент `<Header>` и обязательный элемент `<Body>`. Вот как это выглядит:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
 <soap:Header>
 </soap:Header>
 <soap:Body>
 </soap:Body>
</soap:Envelope>
```

Обратите внимание, что элементы `<Envelope>`, `<Body>` и `<Header>` существуют в пространстве имен SOAP-конверта.

Элемент <Body> содержит полезную нагрузку сообщения. Именно здесь и размещаются фактические данные, такие как параметры (если речь идет о сообщении запроса) или возвращаемое значение (если речь идет об ответном сообщении). Здесь также может содержаться информация о неисправностях в виде условия ошибки или *независимых элементов*, определяющих правила сериализации сложных типов.

Сообщения запросов

В случае, когда речь идет о веб-службах, которые генерировались в .NET автоматически, первый элемент в элементе <Body> — это имя вызываемого метода. Например, вот как выглядит SOAP-сообщение для вызова метода GetEmployeesCount():

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
 <soap:Body>
 <GetEmployeesCount xmlns="http://www.apress.com/ProASP.NET/" />
 </soap:Body>
</soap:Envelope>
```

В этом примере все, что нужно — это пустой элемент <GetEmployeesCount>. Однако если бы данный метод принимал какие-нибудь параметры, они бы были закодированы в элементе <GetEmployeesCount> под соответствующими именами. Например, показанное ниже SOAP-сообщение представляет вызов метода GetEmployeesByCity(), в котором в качестве названия города указывается значение London (Лондон):

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
 <soap:Body>
 <GetEmployeesByCity xmlns="http://www.apress.com/ProASP.NET/">
 <city>London</city>
 </GetEmployeesByCity>
 </soap:Body>
</soap:Envelope>
```

Нетрудно заметить, что в обоих примерах данным внутри элемента <Body> присваивается пространство имен веб-службы (в последнем примере это пространство имен http://www.apress.com/ProASP.NET). Протокол SOAP на самом деле является достаточно гибким и допускает любую XML-разметку в элементе <Body>. Это означает, что вместо того, чтобы рассматривать протокол SOAP просто как протокол для вызова удаленных методов, вы можете также использовать его как способ для обмена сложными XML-документами. В сценарии типа “бизнес для бизнеса” разные части такого документа могут создаваться разными компаниями в автоматизированном рабочем потоке и даже могут включать цифровые подписи. К сожалению, модель программирования .NET превращает работу в таком режиме в достаточно сложную задачу, поскольку она скрывает детали протоколов SOAP и WSDL посредством объектно-ориентированной абстракции. Однако ведущие разработчики веб-служб пытаются сделать такой подход возможным, и вполне вероятно, что будущие версии ASP.NET будут иметь большую степень гибкости.

На заметку! На данном этапе вас наверняка интересует вопрос о том, существует ли вероятность создать несовместимые SOAP-сообщения. Например, веб-службы, разрабатываемые в .NET, для первого элемента в <Body> используют имя метода, но другие реализации веб-служб могут не следовать этому правилу. Проблемы подобного рода решены в стандарте WSDL, который более подробно будет рассматриваться чуть позже в этой главе, в разделе “Язык WSDL”. Он позволяет .NET намного более подробно определять формат сообщений для веб-служб. Другими словами, тот факт, что правила присвоения имен и организации SOAP-сообщения в .NET несколько отличаются от тех, которые используются на конкурирующей платформе, не будет иметь никакого значения, потому что платформа, отличная от .NET, чтобы понять, как ей следует взаимодействовать с данной веб-службой, будет считывать правила, которые перечислены в документе WSDL.

Ответные сообщения

Отправив сообщение с запросом, клиент ожидает получить ответ с веб-сервера (так же, как и в случае с HTTP-запросом). Ответное сообщение имеет практически такой же формат, как и сообщение запроса. Согласно правилам .NET (что, опять-таки, не является требованием SOAP-спецификации), первым дочерним элементом в элементе <Body> является имя вызывавшегося метода, за которым следует суффикс *Response*. Например, вызвав метод GetEmployeesCount(), вы могли бы получить такой ответ:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
 <soap:Body>
 <GetEmployeesCountResponse xmlns="http://www.apress.com/ProASP.NET/">
 <GetEmployeesCountResult>9</GetEmployeesCountResult>
 </GetEmployeesCountResponse>
 </soap:Body>
</soap:Envelope>
```

Как и в сообщении запроса, в ответном сообщении обязательно указывается пространство имен веб-службы. Внутри элемента <GetEmployeesCountResponse> находится элемент, отвечающий за возвращаемое значение, который в данном случае называется <GetEmployeesCountResult>. Согласно правилам .NET, имя этого элемента состоит из имени вызывавшегося метода и суффикса *Result*. Интересно, что это не вся информация, которую можно найти в элементе <XxxREsponse>. Если в методе использовались ссылочные или выходные параметры, здесь также будут содержаться и данные для этих параметров. Это позволяет клиенту обновлять значения параметров после того, как завершится выполнение метода, что дает такое же поведение, как и тогда, когда выходные или ссылочные параметры применяются для вызова локального метода.

Сообщения об ошибках

Стандарт SOAP также предлагает способ для представления сбойных ситуаций. Если на сервере происходит ошибка, клиенту отправляется сообщение, первым элементом в элементе <Body> которого является <Fault>. К счастью, .NET следует этому стандарту и делает это автоматически. Если во время выполнения веб-метода возникает необработанное исключение, .NET отправляет клиенту SOAP-сообщение об ошибке. Когда прокси-класс получает это сообщение об ошибке, он генерирует на стороне клиента исключение, чтобы уведомить клиентское сообщение о произошедшем сбое. Однако, как вы увидите, этот процесс преобразования исключения веб-службы в исключение клиентского приложения проходит не совсем гладко.

Для примера давайте представим, что произойдет, если мы вызовем метод GetEmployeesCount(), когда сервер базы данных недоступен. На веб-сервере будет выдано исключение SqlException. Это исключение перехватит ASP.NET и вернет следующее сообщение об ошибке (которое мы представили здесь в несколько сокращенном виде):

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
 <soap:Body>
 <soap:Fault>
 <faultcode>soap:Server</faultcode>
 <faultstring>System.Web.Services.Protocols.SoapException: Server was unable
to process request. ---> System.Data.SqlClient.SqlException: SQL Server does not
exist or access denied. at ... </faultstring>
 <detail />
 </soap:Fault>
 </soap:Body>
</soap:Envelope>
```

Как правило, элемент Fault всегда содержит такие элементы: <faultcode>, <faultstring> и <detail>. Элемент <faultcode> может принимать одно из предопределенных значений: ClientFaultCode (которое обозначает, что проблема возникла из-за ошибки в посланном клиентом SOAP-запросе), MustUnderstandFaultCode (которое обозначает, что веб-службе не удалось распознать требуемую часть SOAP-сообщения), ServerFaultCode (которое обозначает, что проблема возникла из-за ошибки на сервере) и VersionMismatchFaultCode (которое обозначает, что было обнаружено недействительное пространство имен). Элемент <faultstring> содержит полное описание проблемы. Являющийся необязательным элемент <detail> может использоваться для хранения дополнительной информации о произошедшей ошибке (хотя в данном примере он никаких таких данных не содержит).

Проблема состоит в том, что элемент <Fault> не отображается напрямую на .NET-класс исключений. Когда прокси-класс получает такое сообщение, он не может идентифицировать исходный объект исключения (и не имеет никакой возможности узнать, доступен ли вообще класс этого исключения на стороне клиента). Поэтому прокси-класс просто генерирует исключение общего типа — SoapException, используя в качестве его описания информацию из раздела <faultstring>.

Чтобы понять, как это работает, давайте представим, что произойдет, если мы используем в нашем клиентском приложении такой код:

```
EmployeesService proxy = new EmployeesService();
int count = -1;
try
{
 count = proxy.GetEmployeesCount();
}
catch (SqlException err)
{ ... }
```

В этом случае исключение никогда не будет перехватываться, потому что это исключение SoapException, а не SqlException (несмотря на то, что основной причиной проблемы и исходным объектом исключения все-таки является исключение SqlException). Даже если мы перехватим исключение SqlException в веб-методе и вручную сгенерируем другой объект исключения, он все равно будет преобразован в SoapException на стороне клиента. По этой причине клиенту очень трудно различать разные типы условий ошибки. Клиент может перехватывать только исключение System.Net.WebException (которое обозначает истечение времени тайм-аута или общую сетевую проблему) либо исключение System.Web.Services.Protocols.SoapException (обозначающее любое .NET-исключение, которое произошло в веб-службе).

У нас имеется еще одна возможность. Можно перехватить это исключение в веб-методе на стороне сервера и самостоятельно сгенерировать поддерживаемое исключение SoapException. Преимуществом такого подхода является то, что мы можем сконфигурировать объект SoapException до того, как веб-служба сгенерирует его, вставив дополнительные XML-данные в элемент <detail>. Тогда клиент сможет прочитать это содержимое и использовать его, чтобы программно определить, что же произошло на самом деле.

Например, ниже показана заведомо сбойная версия метода GetEmployeesCount(), в которой такой подход применяется для добавления в объект SoapException наименования исходного типа исключения с помощью специального элемента <ExceptionType>. Вы можете расширить этот подход, добавив любую комбинацию элементов, атрибутов и данных.

```
[WebMethod()]
public int GetEmployeesCountError()
{
 SqlConnection con = null;
 try
 {
   con = new SqlConnection(connectionString);
   // Преднамеренно создаем сбойную SQL-строку.
   string sql = "INVALID_SQL COUNT(*) FROM Employees";
   SqlCommand cmd = new SqlCommand(sql, con);
   con.Open();
   return (int)cmd.ExecuteScalar();
 }
 catch (Exception err)
 {
   // Готовим информацию для добавления в элемент <detail>
   // и элемент <ExceptionType> с названием типа.
   XmlDocument doc = new XmlDocument();
   XmlNode node = doc.CreateNode(XmlNodeType.Element,
      SoapException.DetailElementName.Name,
      SoapException.DetailElementName.Namespace);
   XmlNode child = doc.CreateNode(XmlNodeType.Element,
      "ExceptionType", SoapException.DetailElementName.Namespace);
   child.InnerText = err.GetType().ToString();
   node.AppendChild(child);
   // Создаем специальный объект SoapException.
   // Используем сообщение из исходного исключения
   // и добавляем информацию о деталях.
   SoapException soapErr = new SoapException(err.Message,
      SoapException.ServerFaultCode, Context.Request.Url.AbsoluteUri, node);
   // Выдаем измененное исключение SoapException.
   throw soapErr;
 }
 finally
 {
   con.Close();
 }
}
```

Клиентское приложение сможет считывать элемент <ExceptionType> для извлечения дополнительной информации, которую мы добавили. Ниже показан пример кода, отображающий название исключения в окне сообщений Windows (рис. 36.6):

```
EmployeesService proxy = new EmployeesService();
try
{
  int count = proxy.GetEmployeesCountError();
}
catch (SoapException err)
{
  MessageBox.Show("Original error was: " + err.Detail.InnerText);
}
```

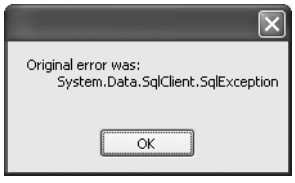

Рис. 36.6. Извлечение дополнительной SOAP-информации о произошедшей ошибке

SOAP-заголовок

SOAP также определят раздел <Header>, в который может помещаться вспомогательная информация. Это, как правило, информация, которая не имеет никакого отношения к полезной нагрузке сообщения. Например, заголовок SOAP-сообщения может содержать данные аутентификационного удостоверения пользователя или идентификатор сеанса. Эти детали могут быть необходимы для обработки запроса, но они не имеют прямого отношения к вызываемому методу. Разделяя эти две части SOAP-сообщения, вы получаете следующие преимущества.

- *Более простой интерфейс метода.* Например, в этом случае вам не придется создавать для метода `GetEmployees()` версию, которая в качестве параметров будет принимать имя пользователя и пароль; эта информация будет передаваться в заголовке, делая данный метод менее запутанным.
- *Более гибкую службу.* Например, если вы добавляете аутентификационную службу и используете SOAP-заголовок, вы сможете изменить способ, которым эта служба работает, и информацию, которую она требует, не изменяя интерфейса веб-методов. При программировании любого типа предпочтение практически всегда отдается слабосвязанным решениям.

Элемент <Header> является необязательным и допускает включение неограниченного числа дочерних элементов. Чтобы определить новые заголовки, которые должны использоваться с веб-службой .NET, потребуется создать классы, унаследованные от класса `System.Web.Services.Protocols.SoapHeader`.

Например, предположим, что вы хотите обеспечить более простой способ поддержки состояния в веб-службе. Вместо того чтобы пытаться использовать cookie-набор с идентификатором сеанса (который предполагает применение cookie-набора HTTP и не может быть определен в документе WSDL), вы могли бы передавать идентификатор сеанса в виде заголовка с каждым SOAP-сообщением. О том, как именно реализуется такой подход, более подробно рассказывается в следующих разделах.

Специальный заголовок

Первое, что нужно сделать, чтобы реализовать такой подход — это создать специальный класс, унаследованный от класса `SoapHeader` и включающий информацию, которая должна передаваться в виде общедоступных свойств.

Например:

```
public class SessionHeader: SoapHeader
{
 public string SessionID;
 public SessionHeader(string sessionID)
 {
   SessionID = sessionID;
 }
```

```
 // Для автоматической десериализации требуется конструктор по умолчанию.
 public SessionHeader()
 {}
}
```

Класс SoapHeader — это на самом деле не более чем простой контейнер данных, который может подвергаться сериализации как в элементе <Header> SOAP-сообщения, так и за его пределами. Специальный класс SessionHeader добавляет строковую переменную SessionID и ключ сеанса.

Связывание заголовка с веб-службой

Чтобы использовать SessionHeader в веб-службе, вам необходимо создать в веб-службе общедоступную переменную экземпляра для этого заголовка:

```
public class SessionHeaderService : System.Web.Services.WebService
{
  public SessionHeader CurrentSessionHeader;
  ...
}
```

При создании прокси-класса для данной веб-службы в него автоматически включается свойство CurrentSessionHeader. Используя это свойство, вы можете считывать или устанавливать заголовок сеанса. Определение для этого специального класса SessionHeader тоже добавляется в файл прокси-класса.

Заголовки связываются с отдельными методами с помощью атрибута SoapHeader. Например, если требуется использовать службу SessionHeader в веб-методе под именем DoSomething(), вы могли бы применить атрибуты WebMethod и SoapHeader следующим образом:

```
[WebMethod()]
[SoapHeader("CurrentSessionHeader")]
public void DoSomething()
{}
```

Обратите внимание, что атрибут SoapHeader в качестве параметра принимает имя общедоступной переменной экземпляра, в которой .NET должен сохранять SOAP-заголовок. В методе DoSomething() атрибут SoapHeader указывает ASP.NET создать новый объект SessionHeader, используя информацию заголовка, которая была получена от клиента, и сохранить ее в общедоступной переменной экземпляра класса CurrentSessionHeader веб-службы. Чтобы найти эту переменную экземпляра во время выполнения, ASP.NET использует рефлексию. Если этой переменной не окажется на месте, произойдет ошибка. Атрибут SoapHeader также может принимать в качестве параметра именованное свойство Direction, которое указывает, как будет отправляться SOAP-заголовок: от клиента веб-службе, из веб-службы веб-клиенту или в обоих направлениях.

Следующий пример демонстрирует как с помощью заголовка сеанса (SessionHeader) можно создать простую систему для сохранения информации о состоянии. Сначала веб-метод CreateSession() позволяет клиенту инициировать новый сеанс. На этом этапе для нового объекта SessionHeader генерируется новый идентификатор сеанса. Далее в коллекции Application создается новая коллекция Hashtable, которая индексируется уже под этим идентификатором сеанса. Идентификатор сеанса использует глобально уникальный идентификатор (GUID), который гарантированно будет уникальным для каждого пользователя.

```
[WebMethod()]
[SoapHeader("CurrentSessionHeader", Direction=SoapHeaderDirection.Out)]
```

```
public void CreateSession()
{
 // Создаем заголовок.
 CurrentSessionHeader = new SessionHeader(Guid.NewGuid().ToString());
 // С этого момента все данные сеанса будут индексироваться под этим ключом.
 Application[CurrentSessionHeader.SessionID] = new Hashtable();
}
```

Эта коллекция `Hashtable` как раз и будет использоваться для хранения дополнительной информации о сеансе. Такой подход нельзя назвать наилучшим (например, коллекция `Application` не используется совместно веб-кластером, не сохраняется в случае перезапуска веб-приложения и не способна обслуживать большое количество пользователей). Однако вы запросто могли бы расширить его, добавив серверную базу данных и кэширование. Такое решение имело бы намного большую масштабируемость и использовало бы ту же систему заголовков сеансов, что и в данном примере.

Вы также заметите, что метод `CreateSession()` использует направление `SoapHeaderDirection.Out`, потому что он создает заголовок и отправляет его обратно клиенту. Здесь присутствует один интересный момент: когда клиент получает специальный заголовок, тот сохраняется в свойстве `CurrentSessionHeader` прокси-класса. С этого момента всякий раз, когда клиентское приложение вызывает метод веб-службы, который требует заголовок, он отправляется вместе с запросом. На самом деле, до тех пор, пока клиент использует один и тот же прокси-класс, заголовки передаются автоматически, и система управления сеансами остается полностью прозрачной.

Чтобы проверить это, вам придется добавить в веб-службу еще два метода. Первый метод — это `SetSessionData()`, который принимает набор данных (`DataSet`) и сохраняет его в сегменте коллекции `Application` для текущего сеанса пользователя.

```
[WebMethod()]
[SoapHeader("CurrentSessionHeader", Direction=SoapHeaderDirection.In)]
public void SetSessionData(DataSet ds)
{
 Hashtable session = (Hashtable)Application[CurrentSessionHeader.SessionID];
 session.Add("DataSet", ds);
}
```

На заметку! Блокировать коллекцию `Application` в этом примере не требуется, потому что вероятность использования двумя клиентами одного и того же идентификатора сеанса исключена, а, значит, исключена и вероятность того, что два каких-нибудь пользователя попытаются изменить этот сегмент коллекции `Application` одновременно.

Далее вы можете воспользоваться методом `GetSessionData()`, чтобы извлечь набор данных (`DataSet`) для текущего сеанса пользователя и вернуть его:

```
[WebMethod()]
[SoapHeader("CurrentSessionHeader", Direction=SoapHeaderDirection.In)]
public DataSet GetSessionData()
{
 Hashtable session = (Hashtable)Application[CurrentSessionHeader.SessionID];
 return (DataSet)session["DataSet"];
}
```

Конечно, если бы вы создавали реальную реализацию этой модели, вы бы не сохраняли информацию сеанса в состоянии приложения, потому что такой подход является ненадежным и не обеспечивает необходимой степени масштабируемости. (Чтобы вспомнить, какие проблемы могут возникать, когда используется состояние сеанса, вернитесь к главе 6.) Вместо этого вы, скорее всего, предпочли бы сохранить эту ин-

формацию в конечной базе данных и поместить ее в кэш данных (см. главу 11), чтобы ускорить процесс ее извлечения.

Использование веб-службы, в которой применяется специальный заголовок

Когда веб-метод требует SOAP-заголовок, способа протестировать его с помощью более простых протоколов HTTP GET или HTTP POST не существует. Поэтому вы не сможете протестировать код на отображаемой в браузере тестовой странице. (На самом деле кнопка Invoke (Вызвать) даже не появится на этой странице.) Вместо этого вам придется создать простое клиентское приложение.

Следующий фрагмент кода демонстрирует пример тестирования. Он создает сеанс (когда получает SOAP-заголовок), сохраняет на сервере новый пустой объект `DataSet` и затем извлекает его.

```
SessionHeaderService proxy = new SessionHeaderService();
proxy.CreateSession();
proxy.SetSessionData(new DataSet("TestDataSet"));
DataSet ds = proxy.GetSessionData();
```

SOAP-сообщение, используемое для вызова метода `CreateSession()`, похоже на те, что приводились в предыдущих примерах:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
 <soap:Body>
 <CreateSession xmlns="http://www.apress.com/ProASP.NET/" />
 </soap:Body>
</soap:Envelope>
```

Ответное сообщение включает SOAP-заголовок:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
 <soap:Header>
 <SessionHeader xmlns="http://tempuri.org/">
 <SessionID>bbc0bfed-c3c2-4552-b70e-dfa5564447fd</SessionID>
 </SessionHeader>
 </soap:Header>
 <soap:Body>
 <CreateSessionResponse xmlns="http://www.apress.com/ProASP.NET/" />
 </soap:Body>
</soap:Envelope>
```

Теперь все последующие вызовы данного метода также будут автоматически включать этот SOAP-заголовок, как показано ниже:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
 <soap:Header>
 <SessionHeader xmlns="http://tempuri.org/">
 <SessionID>bbc0bfed-c3c2-4552-b70e-dfa5564447fd</SessionID>
 </SessionHeader>
 </soap:Header>
 <soap:Body>
 <GetSessionData xmlns="http://www.apress.com/ProASP.NET/" />
 </soap:Body>
</soap:Envelope>
```

В результате вы получаете веб-службу, которая реализует альтернативный механизм управления состояниями сеансов, использующий SOAP-заголовки вместо менее надежных cookie-наборов HTTP. Однако SOAP-заголовки могут применяться и для многих других расширений веб-служб. На самом деле в следующей главе вы увидите, как они

вместе с компонентом Web Services Enhancements (производства Microsoft) позволяют использовать новые и развивающиеся стандарты веб-служб.

Язык WSDL

WSDL (Web Service Description Language — язык описания веб-служб) — это основанный на XML язык, используемый для описания общедоступного интерфейса веб-службы и протоколов связи, которые она поддерживает. WSDL-документ, по сути, представляет собой контракт, в котором содержится вся информация, необходимая клиенту, чтобы взаимодействовать с данной веб-службой. В принципе, WSDL-документ играет ту же роль, что библиотека типов для COM-компонента. (Прямого аналога библиотеке типов в мире .NET не существует, потому что вся необходимая описывающая типы информация помещается в компилируемую сборку в виде метаданных.)

Протокол SOAP предоставляет возможность налаживать связь с веб-службой. Однако о том, как следует форматировать сообщения, он ничего не говорит. Без WSDL вам пришлось бы самостоятельно документировать и объяснять, какой формат ваши веб-службы ожидают увидеть в SOAP-конверте. Установив местонахождение веб-службы, разработчики клиентов должны были бы изучать эту информацию и вручную создавать соответствующие SOAP-сообщения запроса и ответа. Если бы такое вмешательство человека требовалось для получения доступа к каждой новой веб-службе, очевидно, что развитие технологии веб-служб существенно замедлилось бы.

Язык WSDL заполняет этот пробел благодаря тому, что он описывает поддерживаемые протоколы и ожидаемые форматы сообщений, которые использует веб-служба. Мощь стандарта WSDL в том, что он не привязан ни к какой определенной платформе или объектной модели. Это XML-язык, который предоставляет интерфейс для веб-служб на всех платформах.

Полную информацию о стандарте WSDL можно найти по адресу `http://www.w3.org/TR/wsdl`. Этот стандарт является достаточно сложным, но лежащая в его основе логика скрывается от глаз разработчика при программировании веб-служб в ASP.NET во многом подобно тому, как скрываются “рабочие” детали HTML-дескрипторов и атрибутов при создании в ASP.NET веб-элементов управления.

Совет. Вам может быть и не придется просматривать WSDL-документ, что зависит от типа веб-службы, которую вы создаете: например, вы можете решить позволить .NET сгенерировать его автоматически. Однако если вы хотите обеспечить поддержку для сторонних клиентов или планируете использовать технологии разработки на основе контрактов (описываемые в разделе “Реализация существующего контракта”), вы должны хорошо разбираться в том, что собой представляет этот документ.

Просмотр WSDL-документа веб-службы

Создав веб-службу, вы можете запросто заставить ASP.NET сгенерировать для нее соответствующий WSDL-документ. Все, что вам нужно сделать — это запросить `.asmx`-файл веб-службы и добавить в конце URL-адреса параметр `?WSDL`. (Другой вариант — щелкнуть на ссылке Service Description (Описание службы) на отображаемой в браузере тестовой странице, которая запрашивает данный URL-адрес.) На рис. 36.7 показан фрагмент WSDL-документа службы `EmployeesService`, которую мы создавали в предыдущей главе.

Рис. 36.7. WSDL-документ для веб-службы `EmployeesService`

WSDL-документ является очень важным, потому что он позволяет конструкторам сред программирования, таких как .NET, создавать средства, которые могут генерировать прокси-классы программным путем. Когда вы добавляете веб-ссылку в Visual Studio (или используете утилиту `wsdl.exe`), вы указываете ей на WSDL-документ веб-службы. (Если это веб-служба .NET, вы можете сэкономить один шаг, указав ей на `.asmx`-файл веб-службы, поскольку оба средства достаточно интеллектуальны и добавляют в конце строки запроса параметр `?WSDL`, чтобы извлечь WSDL-документ для веб-службы .NET.) Затем средство сканирует WSDL-документ и создает прокси-класс, который использует те же методы, параметры и типы данных. Другие языки и платформы программирования тоже предоставляют средства, которые могут работать подобным образом.

На заметку! WSDL-документ содержит информацию, которая необходима для осуществления обмена данными между веб-службой и клиентом. В нем нет никакой информации о коде или том, как реализуются методы данной веб-службы, которая является излишней и может стать причиной нарушения системы безопасности. Помните, что при добавлении веб-ссылки все, что вам нужно — это WSDL-документ. Ни Visual Studio, ни утилита `wsdl.exe` не позволяют просматривать код веб-службы напрямую.

WSDL-документы, как правило, очень длинные, и их просмотр отнимает намного больше усилий, чем просмотр простого SOAP-сообщения. В следующих разделах для примера будет рассмотрен WSDL-документ веб-службы `EmployeesService`.

Базовая структура

WSDL-документы состоят из пяти основных элементов, которые все вместе описывают веб-службу. Первые три из них являются абстрактными и содержат информацию, касающуюся обмена сообщениями. Два последних — более конкретны и содержат информацию о протоколах и адресе.

Первые три абстрактных элемента — `<types>`, `<messages>` и `<portType>` — все вместе определяют интерфейс веб-службы. Они определяют методы, параметры и свойства веб-службы. Два оставшихся более конкретных элемента — `<binding>` и `<port>` — вместе предоставляют информацию о протоколах (SOAP и HTTP) и адресе (URI) веб-службы. Разделение информации об обмене сообщениями и информации о местоположении и протоколах позволяет многократно использовать общий набор сообщений и типов данных в разных протоколах. Это делает WSDL-документы несколько сложнее, чем они могли бы быть в противном случае, но зато оставляет огромные возможности на будущее. Например, в будущем WSDL-документы, возможно, будут служить для описания правил взаимодействия с веб-службами, использующими протокол SMTP, протокол FTP или какой-нибудь совершенно другой сетевой протокол.

Спецификация WSDL 1.1 поставляется вместе с расширениями SOAP через HTTP, HTTP GET, HTTP POST и MIME, которые накладываются поверх базовой спецификации. ASP.NET поддерживает все эти расширения, за исключением MIME. Поскольку WSDL чаще всего реализуется с расширением SOAP поверх HTTP и поскольку именно это расширение используется в ASP.NET по умолчанию, в данной главе WSDL будет рассматриваться по отношению к этому расширению. Протокол SOAP действительно является доминирующим и, похоже, на сегодняшний день у него нет никаких реальных конкурентов. Что касается Microsoft, то создается впечатление, что она больше заинтересована в поддержке SOAP поверх TCP, т.е. протокола пересылки данных DIME (Direct Internet Message Encapsulation — прямая инкапсуляция сообщений в Интернете), и разрабатывает способы, позволяющие обойти используемую HTTP структуру “запрос-ответ”, подобные тем, что доступны в случае поддержки MIME.

Элемент `<definitions>` является корневым элементом WSDL-документа. Именно здесь определяется большая часть пространств имен. Внутри элемента `<definitions>` находятся пять основных элементов, один из которых — `<message>` — встречается несколько раз.

Вот как выглядит базовая структура WSDL-документа:

```
<?xml version="1.0" encoding="utf-8" ?>
<definitions>
 <types></types>
 <message></message>
 <message></message>
 <portType></portType>
 <binding></binding>
 <service></service>
</definitions>
```

Ниже представлено краткое описание основных элементов WSDL-документа.

- **`<types>`**. В этом разделе определяются все используемые веб-службой типы данных, включая специальные типы данных и форматы сообщений.

- **<message>**. Этот раздел содержит информацию о сообщениях запроса и ответа, используемых для обмена данными с веб-службой.
- **<portType>**. В этом разделе сообщения группируются по парам "входное-выходное". Каждая пара представляет какой-нибудь метод.
- **<binding>**. Этот раздел содержит информацию о транспортных протоколах, которые поддерживает данная веб-служба.
- **<service>**. Этот раздел содержит сведения о конечной точке данной веб-службы (URI-адрес).

Раздел <types>

В разделе <types> определяются все используемые веб-службой типы данных. Элемент <types> — это на самом деле вложенная XML-схема, и все типы данных, определенные в стандарте XML Schema, являются допустимыми. Если необходимо, вы можете добавить другие системы типов за счет расширяемости.

В веб-службе .NET каждое сообщение определено как сложный тип. Определение сложного типа описывает имя метода, его параметры, минимальное и максимальное количество раз, которое он может встречаться, и используемые им типы данных. Например, рассмотрим метод GetEmployeesCount(). Его сообщение запроса не требует никаких данных и определяется с помощью синтаксиса XML-схемы, примерно так:

```
<s:element name="GetEmployeesCount">
 <s:complexType />
</s:element>
```

Сообщение ответа возвращает целое число (тип int из стандарта XML Schema) и определяется следующим образом:

```
<s:element name="GetEmployeesCountResponse">
 <s:complexType>
 <s:sequence>
 <s:element minOccurs="1" maxOccurs="1" name="GetEmployeesCountResult"
 type="s:int" />
 </s:sequence>
 </s:complexType>
</s:element>
```

Раздел <types>, скорее всего, будет достаточно длинным, потому что он определяет два сложных типа для каждого веб-метода.

Более интересный пример — веб-служба, возвращающая специальный объект. Для примера давайте рассмотрим следующую версию метода GetEmployees(), которая возвращает массив объектов EmployeeDetails:

```
[WebMethod()]
public EmployeeDetails[] GetEmployees()
{ ... }
```

Если вы посмотрите на раздел <types> этой веб-службы, то увидите, что сообщение запроса осталось таким же, как было:

```
<s:element name="GetEmployees">
 <s:complexType />
</s:element>
```

Однако сообщение ответа теперь ссылается на другой сложный тип, который называется ArrayOfEmployeeDetails:

```
<s:element name="GetEmployeesResponse">
 <s:complexType>
 <s:sequence>
 <s:element minOccurs="0" maxOccurs="1" name="GetEmployeesResult"
 type="s0:ArrayOfEmployeeDetails" />
 </s:sequence>
 </s:complexType>
</s:element>
```

`ArrayofEmployeeDetails` — это сложный тип, который генерируется автоматически. Он представляет собой список, состоящий из нуля или более объектов `EmployeeDetails`, как показано ниже:

```
<s:complexType name="ArrayOfEmployeeDetails">
 <s:sequence>
 <s:element minOccurs="0" maxOccurs="unbounded" name="EmployeeDetails"
 nillable="true" type="s0:EmployeeDetails" />
 </s:sequence>
</s:complexType>
```

Класс данных `EmployeeDetails` тоже определен как сложный тип в разделе `<types>`. Он состоит из следующих элементов: `EmployeeID`, `FirstName`, `LastName` и `TitleOfCourtesy`.

```
<s:complexType name="EmployeeDetails">
 <s:sequence>
 <s:element minOccurs="1" maxOccurs="1" name="EmployeeID" type="s:int" />
 <s:element minOccurs="0" maxOccurs="1" name="FirstName" type="s:string" />
 <s:element minOccurs="0" maxOccurs="1" name="LastName" type="s:string" />
 <s:element minOccurs="0" maxOccurs="1" name="TitleOfCourtesy"
 type="s:string" />
 </s:sequence>
</s:complexType>
```

Более подробно о том, как сложные типы работают в веб-службах, будет рассказываться чуть позже в этой главе (в разделе “Настройка SOAP-сообщений”).

Еще один элемент, который может встречаться в разделе `<types>` — это определение для SOAP-заголовка (или заголовков). Например, код для тестирования веб-службы с поддержкой состояний, который мы создавали ранее в этой главе, определяет следующий тип для представления данных в заголовке сеанса:

```
<s:element name="SessionHeader" type="s0:SessionHeader" />
<s:complexType name="SessionHeader">
 <s:sequence>
 <s:element minOccurs="0" maxOccurs="1" name="SessionID" type="s:string" />
 </s:sequence>
</s:complexType>
```

Раздел <message>

Сообщения представляют информацию, которой метод веб-службы и клиент обмениваются между собой. Когда у простой веб-службы запрашиваются сведения о показателях котировки акций, ASP.NET отправляет соответствующее сообщение, а веб-служба возвращает уже другое сообщение. Определения для этих сообщений можно найти в разделе `<message>` WSDL-документа. Например:

```
<message name="GetEmployeesCountSoapIn">
 <part name="parameters" element="s0:GetEmployeesCount" />
</message>
<message name="GetEmployeesCountSoapOut">
 <part name="parameters" element="s0:GetEmployeesCountResponse" />
</message>
```

В этом примере ASP.NET создает как сообщение `GetEmployeesCountSoapIn`, так и сообщение `GetEmployeesCountSoapOut`. Присвоение имен — это вопрос, связанный с принятыми соглашениями, главное другое: для ввода (отправки параметров и вызова метода веб-службы) и вывода (извлечения возвращаемого значения из метода веб-службы) обязательно должны создаваться отдельные сообщения.

Данные, используемые в этих сообщениях, определяются на основе информации в разделе `<types>`. Например, для сообщения запроса `GetEmployeesCountSoapIn` применяется сообщение `GetEmployeesCount`, которое в разделе `<types>` определено как пустой сложный тип.

Раздел `<portType>`

Информация в разделе `<portType>` WSDL-документа представляет собой своего рода каталог доступных в данной веб-службе функциональных возможностей. В отличие от рассмотренного только что раздела `<message>`, который содержал независимые элементы для входного и выходного сообщения, операции в этом разделе связаны вместе по схеме "запрос-ответ". Имя операции — это имя метода. Раздел `<portType>`, по сути, представляет собой коллекцию операций, как показано ниже:

```
<portType name="EmployeesServiceSoap">
 <operation name="GetEmployeesCount">
 <documentation>Returns the total number of employees.</documentation>
 <input message="s0:GetEmployeesCountSoapIn" />
 <output message="s0:GetEmployeesCountSoapOut" />
 </operation>
 <operation name="GetEmployees">
 <documentation>Returns the full list of employees.</documentation>
 <input message="s0:GetEmployeesSoapIn" />
 <output message="s0:GetEmployeesSoapOut" />
 </operation>
</portType>
```

Кроме того, в этом разделе также будет присутствовать дескриптор `<documentation>`, представляющий информацию, которая была добавлена через свойство `Description` атрибута `WebMethod`.

На заметку! Всего существует четыре типа операций: односторонние, типа "запрос-ответ", типа "просьба-ответ" и операции типа уведомлений. В текущей спецификации WSDL определены привязки только для односторонних операций и операций типа "запрос-ответ". Для оставшихся двух привязки могут быть определены с помощью соответствующих расширений. Последние два типа операций являются просто обратной версией первых двух; единственное отличие между ними заключается в том, где находится запрашиваемая конечная точка: на получающем или отправляющем конце исходного сообщения. HTTP — это протокол, который поддерживает двустороннюю связь, поэтому односторонние операции будут работать только с протоколом MIME (который не поддерживается ASP.NET) или другим специальным расширением.

Раздел `<binding>`

Элементы раздела `<binding>` связывают абстрактный формат данных с конкретным протоколом, который используется для пересылки данных через Интернет-соединение. Пока что WSDL-документ описывал тип данных, применяемый для различных фрагментов информации, обязательные сообщения, используемые для операции, и структуру каждого сообщения. С помощью элемента `<binding>` WSDL-документ описывает низкоуровневый протокол связи, который может использоваться для взаимодей-

ствия с веб-службой. Он связывает этот элемент с элементом <operation> из раздела <portType>.

Не углубляясь во все детали SOAP-кодировки, мы просто приведем пример, демонстрирующий, как SOAP-связь должна работать с методом GetEmployeesCount() веб-службы EmployeesService:

```
<binding name="EmployeesServiceSoap" type="s0:EmployeesServiceSoap">
 <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
 style="document" />
 <operation name="GetEmployeesCount">
 <soap:operation
 soapAction="http://www.apress.com/ProASP.NET/GetEmployeesCount"
 style="document" />
 <input>
 <soap:body use="literal" />
 </input>
 <output>
 <soap:body use="literal" />
 </output>
</operation>
```

Если метод использует SOAP-заголовок, эта информация добавляется в виде элемента <header>. Для примера давайте возьмем метод CreateSession() (из веб-службы, которая создавалась ранее в этой главе). Метод CreateSession() не требует, чтобы клиент предоставлял заголовок, но сам возвращает его. Поэтому ссылку на заголовок будет содержать только выходное сообщение:

```
<operation name="CreateSession">
 <soap:operation soapAction="http://tempuri.org/CreateSession"
 style="document" />
 <input>
 <soap:body use="literal" />
 </input>
 <output>
 <soap:body use="literal" />
 <soap:header message="s0:CreateSessionSessionHeader"
 part="SessionHeader" use="literal" />
 </output>
</operation>
```

Более того, если веб-служба поддерживает SOAP 1.2, разделов <binding> будет два:

```
<binding name="EmployeesServiceSoap12" type="s0:EmployeesServiceSoap">
...
</binding>
```

Не забывайте, что по умолчанию веб-службы .NET поддерживают протокол SOAP как версии 1.1, так и версии 1.2, но вы легко можете изменить это поведение с помощью конфигурационных файлов, как описывалось ранее в этой главе.

Раздел <service>

В разделе <service> определяются входные точки веб-службы в виде одного или более элементов <port>. Каждый элемент <port> содержит информацию об адресе или URI. Ниже показан пример раздела <service>, взятый из WSDL-документа веб-службы EmployeesService:

```
<service name="EmployeesService">
 <documentation>Retrieve the Northwind Employees</documentation>
 <port name="EmployeesServiceSoap" binding="s0:EmployeesServiceSoap">
```

```
 <soap:address location="http://localhost/Chapter33/EmployeesService.asmx" />
 </port>
</service>
```

Раздел `<service>` также включает элемент `<documentation>`, который содержит свойство `Description` атрибута `WebService`, если для него устанавливалось значение.

Реализация существующего контракта

С самого момента появления веб-служб ведутся споры по поводу того, как же их следует разрабатывать. Некоторые разработчики доказывают, что наилучший подход — это использовать платформы, такие как .NET, которые скрывают лежащие в основе детали. Они хотят иметь дело с высокоуровневой инфраструктурой *удаленных вызовов процедур*. А XML-специалисты доказывают, что вся система должна рассматриваться в рамках *передачи XML-сообщений*. Они считают, что при разработке любого приложения веб-службы первым шагом должно быть создание (вручную!) WSDL-документа.

Как и в большинстве спорных ситуаций, правильное решение, пожалуй, находится где-то посредине. Разработчики приложений, скорее всего, никогда не будут создавать WSDL-контракты вручную: это слишком утомительный и не исключающий ошибок процесс. С другой стороны, разработчики, которым необходимо использовать веб-службы в более масштабных межплатформенных сценариях, вынуждены будут обращать внимание на лежащее в основе XML-представление их сообщений и применять технологии вроде атрибутов сериализации XML (рассматриваются в следующем разделе) для того, чтобы быть абсолютно уверенными в том, что они соответствуют нужной схеме.

Версии .NET 1.*x* совершенно очевидно ориентировались на удаленные вызовы процедур. Это существенно ограничивало возможность разработчиков заглядывать "за кулисы" веб-служб и работать с низкоуровневыми деталями. В версии .NET 2.0 это ограничение было устранено путем добавления поддержки для ориентированных на XML подходов. Одним из таких подходов является разработка *на основе контрактов*.

В сценариях разработки веб-службы, которые приводились до сих пор, первым создавался код веб-службы. Соответствующий WSDL-документ ASP.NET генерировал только по требованию. Однако версия .NET 2.0 позволяет подойти к этому вопросу по-другому. Это значит, что вы можете взять какой-нибудь уже существующий WSDL-документ и загрузить его в утилиту командной строки `wsdl.exe`, чтобы создать базовый скелет веб-службы. Все, что вам нужно — это новый переключатель командной строки `/serverInterface`.

Например, чтобы создать определение класса для веб-службы `EmployeesService`, вы могли бы использовать следующую командную строку:

```
wsdl/serverInterface http://localhost/EmployeesService/EmployeesService.asmx?WSDL
```

Вы получите такой интерфейс:

```
public interface IEmployeesServiceSoap
{
 [WebMethod()]
 DataSet GetEmployees();
}
```

Вы затем можете реализовать этот интерфейс в другом классе, чтобы добавить код веб-службы для метода `GetEmployees()`. Когда создание веб-службы начинается с WSDL-контракта, ее точное соответствие ему гарантировано.

На заметку! На самом деле интерфейс не будет таким уж простым. Для гарантии того, что интерфейс будет *в точности* соответствовать WSDL-контракту, .NET добавляет ряд атрибутов, которые более точно определяют детали, такие как пространства имен, SOAP-кодировка и имена XML-элементов. Это усложняет интерфейс; здесь была показана только его базовая структура.

Этот подход также может быть применен и с веб-службой стороннего поставщика. Например, предположим, что вы решили создать свою собственную версию доступной на сайте `XMethods` веб-службы, которая предоставляет сведения об акциях. Вы хотите быть уверены, что клиенты смогут вызывать ваш веб-метод, не извлекая никакого нового WSDL-документа и не испытывая необходимости в перекомпиляции. Чтобы добиться упомянутого эффекта, вы можете сгенерировать и реализовать точную копию интерфейса исходной службы:

```
wsdl /serverInterface
  http://services.xmethods.net/soap/urn:xmethods-delayed-quotes.wsdl
```

Конечно, чтобы ваша веб-служба на самом деле была совместимой, ваш код должен содержать определенные детали, которые не были описаны в WSDL-документе. Эти детали могут касаться правил синтаксического анализа строк, обработки недействительных данных, обработки исключений и т.д.

Разработка на основе контракта вряд ли заменит более простую модель разработки на основе класса. Однако это очень полезная вещь для разработчиков, которым необходимо привязаться к существующим WSDL-контрактам, особенно если речь идет о межплатформенном сценарии.

Настройка SOAP-сообщений

В большинстве случаев волноваться о деталях SOAP-сериализации будет не нужно. Вас вполне может устраивать вариант создания и использования веб-служб с помощью инфраструктуры, предлагаемой .NET. Однако не исключены и случаи, когда вы можете захотеть расширить свои веб-службы так, чтобы они использовали какие-то специальные типы, или преобразовать имеющиеся типы в определенный XML-формат (для межплатформенной совместимости). В следующих разделах речь пойдет о том, как это сделать.

Сериализация сложных типов данных

Как рассказывалось в предыдущей главе, SOAP-спецификация поддерживает все типы данных, определяемые стандартом XML Schema. Такие типы данных считаются *простыми*. Помимо этого, SOAP также поддерживает *сложные* типы, которые представляют собой структуры, состоящие из ряда простых типов. Эти сложные типы могут применяться для возвращаемого значения веб-метода или в качестве параметров. Однако когда веб-метод требует параметров сложного типа, взаимодействовать с ним можно только с помощью SOAP. Более простые механизмы HTTP GET и HTTP POST работать не будут, и вызвать этот веб-метод на отображаемой в браузере тестовой странице не получится.

Мы уже использовали один сложный тип в наших примерах — `DataSet`. При вызове метода `GetEmployees()` в веб-службе `EmployeesService` .NET возвращает XML-документ, который описывает схему объекта `DataSet` и его содержимое. Ниже показан фрагмент этого ответного SOAP-сообщения:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
 <soap:Body>
 <GetEmployeesResponse xmlns="http://www.apress.com/ProASP.NET/">
 <GetEmployeesResult>
 <xs:schema id="NewDataSet" xmlns=""
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
 <!-- Схема не показана. -->
 </xs:schema>
 <diffgr:diffgram xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
 xmlns:diffgr="urn:schemas-microsoft-com:xml-diffgram-v1">
 <EmployeesDataSet xmlns="">
 <Employees diffgr:id="Employees1" msdata:rowOrder="0">
 <EmployeeID>1</EmployeeID>
 <LastName>Davolio</LastName>
 <FirstName>Nancy</FirstName>
 <Title>Sales Representative</Title>
 <TitleOfCourtesy>Ms.</TitleOfCourtesy>
 <HomePhone>(206) 555-9857</HomePhone>
 </Employees>
 <Employees diffgr:id="Employees2" msdata:rowOrder="1">
 <EmployeeID>2</EmployeeID>
 <LastName>Fuller</LastName>
 <FirstName>Andrew</FirstName>
 <Title>Vice President, Sales</Title>
 <TitleOfCourtesy>Dr.</TitleOfCourtesy>
 <HomePhone>(206) 555-9482</HomePhone>
 </Employees>
 ...
 </EmployeesDataSet></diffgr:diffgram>
 </GetEmployeesResult>
 </GetEmployeesResponse>
 </soap:Body>
</soap:Envelope>
```

Вы также можете использовать с веб-службами .NET свои собственные специальные классы. В таком случае при создании прокси в клиентское приложение (на соответствующем ему языке) будет автоматически добавлена копия специального класса.

Процесс преобразования объектов в XML называется *сериализацией*, а процесс воссоздания объектов из XML — *десериализацией*. Класс `System.Xml.Serialization.XmlSerializer` — это компонент, который выполняет сериализацию. Не путайте этот класс с классами сериализации, которые описывались в главе 12, такими как `BinaryFormatter` и `SoapFormatter`. Эти классы выполняют .NET-сериализацию, которая работает, только когда речь идет о патентованных .NET-объектах с атрибутом `Serializable`. В отличие от `BinaryFormatter` и `SoapFormatter`, компонент `XmlSerializer` работает с любым классом, но обладает намного более ограниченными возможностями, нежели класс `BinaryFormatter` и класс `SoapFormatter`, и может извлекать только общедоступные данные.

Чтобы использовать `XmlSerializer` и отправлять специальные объекты в и из веб-службы, вы должны знать о следующих нескольких ограничениях.

- *Любой включаемый код в клиентском приложении игнорируется.* Это означает, что клиентская копия специального класса не будет включать методов, логики для конструктора или логики для процедур свойств. Все эти детали будут автоматически удалены.

- *Класс обязательно должен включать используемый по умолчанию конструктор без аргументов.* Это позволяет .NET создавать новый экземпляр данного объекта при десериализации SOAP-сообщения, содержащего соответствующие данные.
- *Свойства "только для чтения" не подвергаются сериализации.* Другими словами, если свойство предоставляет возможность только для извлечения значения и не предоставляет возможности для установки значения, оно не может быть подвергнуто сериализации. Это также касается индивидуальных свойств и индивидуальных переменных экземпляра.

Очевидно, что необходимость сериализировать класс в часть межплатформенного XML-кода налагает некоторые очень строгие ограничения. Если вы используете специальные классы в веб-службе, лучше относитесь к ним как к простым контейнерам данных, чем как к реальным участникам процесса объектно-ориентированного проектирования.

Создание специального класса

Чтобы посмотреть на класс `XmlSerializer` в действии, придется создать специальный класс и использующий его веб-метод. В следующем примере мы воспользуемся компонентом базы данных, который был разработан еще в главе 8. Этот компонент не использует разрозненных объектов `DataSet`. Вместо этого он возвращает результаты запроса с помощью специального класса `EmployeeDetails`.

Вот как этот класс выглядит на текущий момент, безо всяких связанных с веб-службами улучшений:

```
public class EmployeeDetails
{
 private int employeeID;
 public int EmployeeID
 {
  get {return employeeID;}
  set {employeeID = value;}
 }
 private string firstName;
 public string FirstName
 {
  get {return firstName;}
  set {firstName = value;}
 }
 private string lastName;
 public string LastName
 {
  get {return lastName;}
  set {lastName = value;}
 }
 private string titleOfCourtesy;
 public string TitleOfCourtesy
 {
  get {return titleOfCourtesy;}
  set {titleOfCourtesy = value;}
 }
 public EmployeeDetails(int employeeID, string firstName, string lastName,
    string titleOfCourtesy)
 {
  this.employeeID = employeeID;
  this.firstName = firstName;
  this.lastName = lastName;
  this.titleOfCourtesy = titleOfCourtesy;
 }
}
```

Вместо общедоступных переменных экземпляра в классе `EmployeeDetails` используются процедуры свойств. Однако его все равно можно считать подходящим, потому что класс `XmlSerializer` выполнит необходимое преобразование автоматически. Кроме того, в классе `EmployeeDetails` нет используемого по умолчанию конструктора без параметров, поэтому, прежде чем его можно будет применять в веб-методе, в него обязательно должен быть добавлен приведенный ниже конструктор, например, так:

```
public EmployeeDetails(){}
```

Теперь класс `EmployeeDetails` готов для сценария с веб-службой. Чтобы испробовать его, вы можете создать веб-метод, возвращающий массив объектов `EmployeeDetail`. В следующем примере как раз показан один такой метод, а именно — `GetEmployees()`, который вызывает метод `EmployeeDB.GetEmployees()` в компоненте базы данных. (Полную версию кода для этого метода можно найти в главе 8 или загрузить из сайта издательства.)

Вот как выглядит этот веб-метод:

```
[WebMethod()]
public EmployeeDetails[] GetEmployees()
{
  EmployeeDB db = new EmployeeDB();
  return db.GetEmployees();
}
```

Генерирование прокси

Генерируя прокси (либо с помощью утилиты `wsdl.exe`, либо путем добавления веб-ссылки), вы получите целых два класса. Первый класс — это прокси-класс, используемый для связи с веб-службой. Второй класс — это определение для класса `EmployeeDetails`.

Важно понимать, что клиентская версия класса `EmployeeDetails` отличается от версии, которая находится на стороне сервера. На самом деле у клиента даже нет возможности увидеть весь код класса `EmployeeDetails` на стороне сервера. Вместо этого клиент считывает WSDL-документ, который содержит XML-схему класса `EmployeeDetails`. В этой схеме просто перечисляются все общедоступные свойства и поля (причем они никак не различаются) и их типы данных.

Когда клиент создает прокси-класс, .NET использует информацию из этого WSDL-документа для генерации клиентской версии класса `EmployeeDetails`. Для каждого общедоступного свойства или поля, которое присутствует в определении класса `EmployeeDetails` на стороне сервера, .NET добавляет соответствующее общедоступное свойство в клиентскую версию этого класса.

Вот как выглядит код, сгенерированный для клиентской версии класса `EmployeeDetails`:

```
public partial class EmployeeDetails
{
 private int employeeIDField;
 private string firstNameField;
 private string lastNameField;
 private string titleOfCourtesyField;
 public int EmployeeID
 {
  get { return this.employeeIDField; }
  set { this.employeeIDField = value; }
 }
```

```
 public string FirstName
 {
  get { return this.firstNameField; }
  set { this.firstNameField = value; }
 }
 public string LastName
 {
  get { return this.lastNameField; }
  set { this.lastNameField = value; }
 }
 public string TitleOfCourtesy
 {
  get { return this.titleOfCourtesyField; }
  set { this.titleOfCourtesyField = value; }
 }
}
```

В данном примере клиентская версия очень похожа на версию на стороне сервера, потому что версия на стороне сервера не включала слишком большой объем кода. Единственным явным отличием (помимо переименования индивидуальных полей) является отсутствие конструктора не по умолчанию. Как правило, в клиентской версии не сохраняются ни какие-либо нестандартные конструкторы, ни код в процедурах или конструкторах свойств, ни какие-либо методы, ни какие-либо индивидуальные переменные экземпляра.

На заметку! В клиентской версии класса данных всегда применяются процедуры свойств, даже если в исходной версии этого класса, которая расположена на сервере, используются индивидуальные переменные экземпляра. Это дает нам возможность привязать коллекции клиентских объектов класса `EmployeeDetails` к элементу управлении типа сетки. Подобное поведение было недоступно в версиях .NET 1.*x*.

Тестирование веб-службы со специальным классом

Следующий шаг — написать код, вызывающий метод `GetEmployees()`. Поскольку у клиента теперь есть определение класса `EmployeeDetails`, сделать это несложно:

```
EmployeesServiceCustomDataClass proxy = new EmployeesServiceCustomDataClass();
EmployeeDetails[] employees = proxy.GetEmployees();
```

Ответное сообщение будет включать элемент `<GetEmployeesResult>`, содержащий данные о сотрудниках. По умолчанию `XmlSerializer` также создаст структуру дочерних элементов, основываясь на имени класса (`EmployeeDetails`) и именах общедоступных свойств и переменных (`EmployeeID`, `FirstName`, `LastName`, `TitleOfCourtesy` и т.д.). Интересно, что эта структура очень похожа на XML-код, который мы использовали при создании модели для объекта `DataSet`, без информации о схеме.

Ниже показана сокращенная версия этого ответного сообщения:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
 <soap:Body>
 <GetEmployeesResponse xmlns="http://www.apress.com/ProASP.NET/">
 <GetEmployeesResult>
 <EmployeeDetails>
 <EmployeeID>1</EmployeeID>
 <FirstName>Nancy</FirstName>
 <LastName>Davolio</LastName>
 <TitleOfCourtesy>Ms.</TitleOfCourtesy>
 </EmployeeDetails>
```

```
<EmployeeDetails>
<EmployeeID>2</EmployeeID>
<FirstName>Andrew</FirstName>
<LastName>Fuller</LastName>
<TitleOfCourtesy>Dr.</TitleOfCourtesy>
</EmployeeDetails>
</GetEmployeesResult>
...
</GetEmployeesResponse>
</soap:Body>
</soap:Envelope>
```

Когда клиент получает это сообщение, XML-ответ с помощью клиентской версии определения класса `EmployeeDetails` преобразуется в массив объектов `EmployeeDetails`.

Настройка процесса сериализации XML с помощью атрибутов

В некоторых случаях может возникнуть необходимость настроить XML-представление специального класса. Такой подход наиболее полезен в сценариях межплатформенного программирования, в которых клиент ожидает получить XML в определенном виде. Например, у вас уже может существовать некоторая схема, ожидающая, что в классе `EmployeeDetails` вместо вложенного дескриптора `<EmployeeID>` будет применяться атрибут `EmployeeID`. Каркас .NET предлагает простой способ применить эти правила с помощью атрибутов. Основная идея состоит в следующем: вы применяете атрибуты к своим классам данных (таким как класс `EmployeeDetails`). Когда `XmlSerializer` создает SOAP-сообщение, он считывает эти атрибуты и использует их для корректирования XML-кода, который генерирует.

Пространство `System.Xml.Serialization` содержит целый ряд атрибутов, которые могут использоваться для настройки внешнего вида XML. Существуют два набора атрибутов: в одном наборе атрибуты называются `Xml`*Xxx*, а в другом — `Soap`*Xxx*. Выбор атрибутов зависит от типа используемой для параметров кодировки.

Как уже упоминалось ранее в этой главе, доступны два типа SOAP-кодировки — литеральная и соответствующая требованиям, перечисленным в пятом разделе SOAP-спецификации. Атрибуты `Xml`*Xxx* подходят, когда используются параметры с литеральной кодировкой. Поэтому они применяются в следующих случаях.

- Когда используется веб-служба с кодировкой по умолчанию (т.е., когда никакие атрибуты, изменяющие кодировку, не добавлялись).
- Когда для связи с веб-службой применяются протоколы HTTP GET или HTTP POST.
- Когда атрибут `SoapDocumentService` или `SoapDocumentMethod` используется вместе со свойством `Use`, для которого устанавливается значение `SoapBindingUse.Literal`.
- Когда класс `XmlSerializer` используется сам по себе (за пределами веб-службы).

А атрибуты `Soap`*Xxx* подходят, когда используются параметры с кодировкой согласно требованиям пятого раздела SOAP-спецификации. Поэтому они применяются в перечисленных ниже случаях.

- Когда используются атрибуты `SoapRpcService` или `SoapRpcMethod`.
- Когда атрибут `SoapDocumentService` или `SoapDocumentMethod` используется вместе со свойством `Use`, для которого устанавливается значение `SoapBindingUse.Encoded`.

К члену класса могут одновременно применяться как атрибуты `Soap`*Xxx*, так и атрибуты `Xml`*Xxx*. То, какие из них будут использоваться, зависит от типа выполняемой сериализации.

В табл. 36.1 перечислены практически все доступные атрибуты. Многие атрибуты содержат свойства. Некоторые свойства являются общими для большинства атрибутов, например, такие как свойство `Namespace` (используемое для указания пространства имен подвергаемых сериализации XML-данных) и свойство `DataType` (используемое для указания специфического типа данных XML Schema, который компонент `XmlSerializer` вряд ли выберет по умолчанию). Полное описание всех атрибутов и их свойств можно найти на веб-сайте MSDN, в разделе "Help" ("Справка").

Таблица 36.1. Атрибуты для настройки сериализации XML-данных

Xml-атрибут	SOAP-атрибут	Описание
`XmlAttribute`	`SoapAttribute`	Позволяет сделать так, чтобы поля или свойства преобразовывались не в элементы, а в XML-атрибуты.
`XmlElement`	`SoapElement`	Используется для указания XML-элементов.
`XmlArray`		Используется для указания массивов.
`XmlIgnore`	`SoapIgnore`	Позволяет сделать так, чтобы поля или свойства не подвергались сериализации.
`XmlInclude`	`SoapInclude`	Используется в сценариях наследования. Например, у вас может быть свойство или поле, типизированное как базовый класс, но на самом деле ссылающееся на какой-нибудь порожденный класс. В таком случае вы можете применить атрибут `XmlInclude` для указания всех типов порожденного класса, которые могут использоваться.
`XmlRoot`		Используется для указания элемента наивысшего уровня.
`XmlText`		Позволяет выполнять сериализацию полей прямо в XML-тексте без элементов.
`XmlEnum`	`SoapEnum`	Позволяет присваивать элементам перечисления имена, отличные от тех, что используются для них в этом перечислении.
`XmlType`	`SoapType`	Позволяет изменять названия типов в WSDL-файле.

Чтобы посмотреть, как работает SOAP-сериализация, вы можете применить все эти атрибуты к классу `EmployeeDetails`. Например, ниже показано измененное объявление этого класса, в котором теперь используются несколько атрибутов сериализации:

```
public class EmployeeDetails
{
 [XmlAttribute("id")]
 public int EmployeeID
 {
  get {return employeeID;}
  set {employeeID = value;}
 }
 [XmlElement("First")]
 public string FirstName
 {
  get {return firstName;}
```

```
  set {firstName = value;}
 }
 [XmlElement("Last")]
 public string LastName
 {
  get {return lastName;}
  set {lastName = value;}
 }
 [XmlIgnore()]
 public string TitleOfCourtesy
 {
  get {return titleOfCourtesy;}
  set {titleOfCourtesy = value;}
 }
 // (Конструкторы и приватные данные не показаны.)
}
```

Вот как этот подвергшийся сериализации класс `EmployeeDetails` будет выглядеть в SOAP-сообщении.

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
 <soap:Body>
 <GetEmployeesResponse xmlns="http://www.apress.com/ProASP.NET/">
 <GetEmployeesResult>
 <EmployeeDetails id="1">
 <First>Nancy</First>
 <Last>Davolio</Last>
 </EmployeeDetails>
 <EmployeeDetails id="2">
 <First>Andrew</First>
 <Last>Fuller</Last>
 </EmployeeDetails>
 ...
 </GetEmployeesResult>
 </GetEmployeesResponse>
 </soap:Body>
</soap:Envelope>
```

Совет. При желании поэкспериментировать с различными атрибутами сериализации, вы также можете воспользоваться непосредственно самим классом `XmlSerializer`. Просто создайте экземпляр класса `XmlSerializer` и передайте тип объекта, который хотите подвергнуть сериализации, в виде параметра конструктора. Затем вы можете воспользоваться методом `Serialize()`, чтобы преобразовать этот объект в XML и записать данные в поток или объект `TextWriter`. Чтобы прочитать XML-данные из потока или объекта `TextReader` и воссоздать исходный объект, используйте метод `Deserialize()`. Вы также можете применить утилиту командной строки `xsd.exe`, которая поставляется вместе с .NET Framework, и сгенерировать C#-определения этого класса на основе документов схем XML. В этом случае объявление класса будет автоматически включать все необходимые атрибуты сериализации.

Этот пример имеет только одно ограничение. Несмотря на то что вы можете управлять процессом сериализации объекта `EmployeeDetails`, вы не можете использовать те же самые атрибуты для придания формы элементу, содержащему список сотрудников. Чтобы сделать это, у вас есть два варианта. Вы можете создать специальный класс типа коллекции и применить атрибуты XML-сериализации к нему. Или, если вы хотите продолжать использовать обычный массив, то должны добавить XML-атрибут, применяемый непосредственно к возвращаемому значению веб-метода, например, так:

```
[return: XmlArray("EmployeeList")]
public EmployeeDetails[] GetEmployees()
{ ... }
```

Теперь, вызвав веб-метод, вы получите следующие XML-данные:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
 <soap:Body>
 <GetEmployeesResponse xmlns="http://www.apress.com/ProASP.NET/">
 <EmployeeList>
 <EmployeeDetails id="1">
 <First>Nancy</First>
 <Last>Davolio</Last>
 </EmployeeDetails>
 <EmployeeDetails id="2">
 <First>Andrew</First>
 <Last>Fuller</Last>
 </EmployeeDetails>
 ...
 </EmployeeList>
 </GetEmployeesResponse>
 </soap:Body>
</soap:Envelope>
```

Вы можете сделать еще много чего, чтобы сконфигурировать эти детали. Например, вы можете вставить атрибуты XML-сериализации непосредственно перед своими параметрами, чтобы изменить обязательные XML-данные входящего сообщения запроса. Вы также можете воспользоваться атрибутом `SoapDocument` (который рассматривался ранее в этой главе) и изменить имя и пространство имен XML-элемента, который содержит возвращаемое значение вашей функции (в данном примере он называется `<GetEmployeesResponse>`).

Совместное использование типов

В версиях .NET 1.*x* избежать проблем, когда один и тот же специальный класс используется более одной веб-службой, очень трудно. Например, представьте такую ситуацию: вы вызываете метод `Store.GetOrder()` из одной веб-службы, чтобы извлечь объект `Order`, и затем отправляете этот объект `Order` методу `Shipping.TrackOrder()` из другой веб-службы. Проблема состоит в том, что когда вы добавляете ссылки на обе веб-службы (на веб-службу `Store` и веб-службу `Shipping`), вы в конечном итоге получаете две копии класса данных объекта `Order`, которые находятся в двух разных пространствах имен. И, хотя эти определения класса являются эквивалентными, они не могут использоваться как взаимозаменяемые. Это означает, что, получив версию объекта из одной веб-службы, вы не можете передать ее дальше в другую веб-службу.

В .NET 2.0 эта проблема решается с помощью новой функциональной возможности, которая называется совместным использованием типов. С помощью этой функциональной возможности вы можете сгенерировать одну клиентскую копию класса данных и применять ее со всеми совместимыми веб-службами. Чтобы считаться совместимым, специальный класс данных должен отвечать перечисленным ниже требованиям.

- Он должен иметь такое же XML-представление. Другими словами, XML-схема типа должна быть идентичной.
- Он должен относиться к тому же пространству имен XML. Другие пространства имен указывают на другие документы.

Все остальные детали не являются важными. Например, следующие факторы никак не влияют на возможность выполнять совместное использование типов.

- Местоположение веб-службы.
- Используемый веб-службой язык.
- Имя класса и имена свойств (если применяются атрибуты XML-сериализации, гарантирующие соответствие сериализованной версии).

Например, следующий серверный класс `Employee` существенно отличается от класса `EmployeeDetails`, который приводился в предыдущем разделе, но его сериализованная версия выглядит абсолютно так же:

```
[XmlElement("EmployeeDetails")]
public class Employee
{
 [XmlAttribute("id")]
 public int ID;
 [XmlElement("First")]
 public string FirstName;
 [XmlElement("Last")]
 public string LastName;
}
```

Этот класс отвечает первому требованию (которое гласит о том, что XML-схема должна быть идентичной), но, возможно, не отвечает второму (гласящему о том, что пространство имен тоже должно совпадать). Существуют два способа добиться того, чтобы класс находился в таком же пространстве имен. Первый — это использовать одно и то же пространство имен в атрибуте `WebService` для обеих веб-служб:

```
[WebService(Namespace="http://www.apress.com/ProASP.NET/")
public class EmployeesServiceCompatible : System.Web.Services.WebService
{ ... }
```

Обычно такой подход — не то, что нужно. Он предполагает, что обе веб-службы являются одинаковыми. Более подходящий вариант — использовать уникальное пространство имен и идентифицировать совместно используемые структуры XML-данных. В данном примере это можно сделать, применив к классам `EmployeeDetails` и `Employee` атрибут `XmlRoot` или `XmlElement`, как показано ниже:

```
[XmlElement("EmployeeDetails",
 Namespace="http://www.apress.com/ProASP.NET/EmployeeDetails")]
public class Employee
{ ... }
```

Теперь сериализованные XML-данные в ответном сообщении будут выглядеть так:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
 <soap:Body>
 <GetEmployeesResponse xmlns="http://www.apress.com/ProASP.NET/">
 <GetEmployeesResult>
 <EmployeeDetails id="1"
 xmlns="http://www.apress.com/ProASP.NET/EmployeeDetails">
 <First>Nancy</First>
 <Last>Davolio</Last>
 </EmployeeDetails>
 <EmployeeDetails id="2"
 xmlns="http://www.apress.com/ProASP.NET/EmployeeDetails">
 <First>Andrew</First>
 <Last>Fuller</Last>
 </EmployeeDetails>
 ...
 </GetEmployeesResponse>
 </soap:Body>
</soap:Envelope>
```

Удовлетворив два указанных обязательных требования, можно приступать к генерированию совместно используемого типа. На данный момент это возможно только путем вызова утилиты командной строки `wsdl.exe` с переключателем `/sharetypes`. Также обязательно следует указать адреса всех служб, которые используют один и тот же тип. Например:

```
wsdl /sharetypes
http://localhost/EmployeesService2/EmployeesServiceCompatible.asmx?WSDL
http://localhost/EmployeesService2/EmployeesService.asmx?WSDL
```

Если по какой-то причине типы окажутся неидентичными, соответствующего предупреждающего сообщения не последует. Вместо этого прокси-класс будет содержать множество версий этого класса данных, например, `EmployeeDetails`, `EmployeeDetails1` и т.д.

Настройка процесса сериализации XML с помощью `IXmlSerializable`

Атрибуты XML-сериализации работают хорошо, когда свойства отображаются на XML-элементы или атрибуты по схеме "один к одному". Однако в некоторых сценариях разработчикам необходима большая степень гибкости, чтобы создать XML-представление конкретного типа, отвечающего определенной схеме. Например, им может потребоваться внести какие-нибудь изменения в представление типов данных, изменить порядок элементов или добавить какую-нибудь вспомогательную информацию (типа комментариев или даты сериализации данного документа). В некоторых случаях применение атрибутов XML-сериализации вполне возможно с технической точки зрения, но может подразумевать создание чрезмерно сложной модели класса.

К счастью, .NET 2.0 предоставляет интерфейс `IXmlSerializable`, который вы можете реализовать, чтобы получить полный контроль над своими XML-данными. Атрибут `IXmlSerializable` существовал в .NET, начиная с версии 1.0. Однако он применялся только как патентованный способ для настройки .NET-объекта `DataSet` и не был доступен для общего использования. Теперь он поддерживается полностью. `IXmlSerializable` предоставляет три метода, которые перечислены в табл. 36.2.

Таблица 36.2. Методы атрибута `IXmlSerializable`

Метод	Описание
`WriteXml()`	Этот метод выполняет запись XML-представления экземпляра нужного объекта с помощью класса `XmlWriter`. Он необходим в веб-службе для того, чтобы веб-служба выполняла сериализацию объекта и отправляла его в виде возвращаемого значения.
`ReadXml()`	Этот метод выполняет считывание XML-данных из класса `XmlReader` и генерирует соответствующий объект. Вполне возможно, что этот метод вам и не понадобится (на такой случай не помешает предусмотреть исключение `NotImplementedException`). Однако он обязательно потребуется, если у вас будет необходимость в десериализации объекта, который ваша веб-служба принимает в качестве входного параметра или если вы решите реализовать этот специальный класс на стороне клиента.
`GetSchema()`	Этот метод устарел и должен возвращать значение `null`. Если вы хотите иметь возможность сгенерировать XML-схему для своего класса (которая будет включена в WSDL-документ), вы обязательно должны использовать вместо него атрибут `XmlSchemaProvider`. Атрибут `XmlSchemaProvider` обозначает в вашем классе метод, который возвращает документ со схемой XML-данных (XSD).

Классы `XmlReader` и `XmlWriter` подробно рассматривались в главе 14. Использовать их совсем не сложно. Ниже показан пример специального класса, который самостоятельно поддерживает процесс генерации своих XML-данных:

```
public class EmployeeDetailsCustom : IXmlSerializable
{
 public int ID;
 public string FirstName;
 public string LastName;
 const string ns = "http://www.apress.com/ProASP.NET/CustomEmployeeDetails";
 void IXmlSerializable.WriteXml(XmlWriter w)
 {
   w.WriteStartElement("Employee", ns);
   w.WriteStartElement("Name", ns);
   w.WriteElementString("First", ns, FirstName);
   w.WriteElementString("Last", ns, LastName);
   w.WriteEndElement();
   w.WriteElementString("ID", ns, ID.ToString());
   w.WriteEndElement();
 }
 void IXmlSerializable.ReadXml(XmlReader r)
 {
   r.MoveToContent();
   r.ReadStartElement("Employee");
   r.ReadStartElement("Name");
   FirstName = r.ReadElementString("First", ns);
   LastName = r.ReadElementString("Last", ns);
   r.ReadEndElement();
   r.MoveToContent();
   ID = Int32.Parse(r.ReadElementString("ID", ns));
   reader.ReadEndElement();
 }
 System.Xml.Schema.XmlSchema IXmlSerializable.GetSchema()
 {
   return null;
 }
 // (Конструкторы были опущены.)
}
```

Совет. Удостоверьтесь в том, что считываете весь XML-документ, включая закрывающие дескрипторы элементов в методе `ReadXml()`. В противном случае .NET может сгенерировать исключение, когда вы попытаетесь выполнить десериализацию XML-данных.

Теперь, если вы создадите такой веб-метод:

```
[WebMethod()]
public EmployeeDetailsCustom GetCustomEmployee()
{
 return new EmployeeDetailsCustom(101, "Joe", "Dabiak");
}
```

вот какие XML-данные вы увидите:

```
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
 <soap:Body>
 <GetCustomEmployeeResponse xmlns="http://www.apress.com/ProASP.NET/">
 <GetCustomEmployeeResult>
 <Employee xmlns="http://www.apress.com/ProASP.NET/CustomEmployeeDetails">
```

```
 <Name>
 <First>Joe</First>
 <Last>Tester</Last>
 </Name>
 <ID>1</ID>
 </Employee>
 </GetCustomEmployeeResult>
 </GetCustomEmployeeResponse>
 </soap:Body>
</soap:Envelope>
```

На заметку! Когда используется интерфейс `IXmlSerializable`, единственными атрибутами, которые имеют хоть какой-нибудь эффект, являются те, что применяются к методу или объявлению класса. Атрибуты, применяемые к отдельным свойствам и полям, не имеют никакого эффекта. Однако вы можете воспользоваться рефлексией .NET, чтобы отыскать свои собственные атрибуты, и затем использовать их для настройки генерируемой XML-схемы.

Схемы для специальных типов данных

Единственным ограничением в рассматриваемом примере является то, что у клиента нет никакой возможности определить, какие XML-данные ему следует ожидать. Если вы посмотрите на раздел `<types>` WSDL-документа для этого примера, то увидите, что схема оставлена открытой с помощью элемента `<any>`. Это означает, что далее могут следовать любые допустимые XML-данные.

```
<s:element name="GetCustomEmployeeResponse">
 <s:complexType>
 <s:sequence>
 <s:element minOccurs="0" maxOccurs="1" name="GetCustomEmployeeResult">
 <s:complexType>
 <s:sequence>
 <s:element ref="s:schema" />
 <s:any />
 </s:sequence>
 </s:complexType>
 </s:element>
 </s:sequence>
 </s:complexType>
</s:element>
```

На стороне клиента вы могли бы отнестись к этим данным как к XML-фрагменту и написать код, выполняющий синтаксический анализ XML. Однако предоставить XML-схему для своего специального XML-представления — куда более лучший подход.

Чтобы сделать это, вы должны добавить в свой класс статический метод, который будет возвращать документ XML-схемы в виде объекта `XmlQualifiedName`, как показано ниже:

```
public static XmlQualifiedName GetSchemaDocument(XmlSchemaSet xs)
{
 // Извлекаем путь к файлу схемы.
 string schemaPath = HttpContext.Current.Server.MapPath("EmployeeDetails.xsd");
 // Извлекаем схему из файла.
 XmlSerializer schemaSerializer = new XmlSerializer(typeof(XmlSchema));
 XmlSchema s = (XmlSchema)schemaSerializer.Deserialize(
   new XmlTextReader(schemaPath), null);
 xs.XmlResolver = new XmlUrlResolver();
 xs.Add(s);
 return new XmlQualifiedName("EmployeeDetails", ns);
}
```

Совет. В данном примере документ схемы извлекается из файла. Для достижения более высокой производительности этот документ следовало бы поместить в кэш или создать его программно.

Далее вы должны указать .NET на правильный метод с помощью атрибута `XmlSchemaProvider`:

```
[XmlSchemaProvider("GetSchemaDocument")]
public class EmployeeDetailsCustom : IXmlSerializable
{ ... }
```

Теперь при генерации WSDL-документа ASP.NET вызовет этот статический метод и затем добавит в WSDL-документ информацию о схеме. Однако не следует забывать о том, что при создании клиента .NET сгенерирует класс данных уже согласно данной схеме, что означает, что версия класса `EmployeeDetails` на стороне клиента будет значительно отличаться от его версии на стороне сервера. (В данном примере клиентская версия класса `EmployeeDetails` из-за указанной в схеме организации XML-элементов будет включать вложенный класс `Name`, что, возможно, вам не подходит.)

Итак, что же вы можете сделать, если хотите иметь одинаковую версию класса `EmployeeDetails` на обеих сторонах (т.е. на стороне клиента и на стороне сервера). Вы могли бы вручную изменить сгенерированный код прокси-класса, однако такое изменение придется вносить заново при каждом обновлении прокси. Более надежный (долгосрочный) вариант — использовать расширения импортера схем, но об этом речь пойдет уже в разделе “Расширения импортера схем”.

Специальная сериализация для больших типов данных

Одной из причин, по которой может быть принято решение использовать интерфейс `IXmlSerializable`, является необходимость создать веб-службу, способную пересылать большие объемы данных. Например, предположим, что вам необходимо переслать большой блок двоичных данных, который включает содержимое из файла. Вы могли бы воспользоваться такой веб-службой:

```
[WebMethod()]
public byte[] DownloadFile(string fileName)
{ ... }
```

Проблема состоит в том, что такой подход предполагает, что в память (в виде байтового массива) будут считываться сразу все данные файла. Если размер файла составляет несколько гигабайт, это может привести к зависанию компьютера. Более удачное решение — воспользоваться атрибутом `IXmlSerializable` и реализовать разбивку данных на фрагменты. Таким образом, вы сможете пересылать любое количество данных, записывая их по одному фрагменту за раз.

В следующих разделах будет показан пример, в котором использование интерфейса `IXmlSerializable` позволяет значительно снизить количество непроизводительных издержек при пересылке файла больших размеров.

Серверная сторона

Первое, что нужно сделать — это создать сигнатуру для веб-метода. Чтобы эта стратегия работала, веб-метод должен возвращать класс, реализующий интерфейс `IXmlSerializable`. В приведенном ниже примере для этого используется класс под названием `FileData`. Помимо этого, еще потребуется отключить ASP.NET-функцию буферизации, чтобы ответ мог поступать через сеть.

```
[WebMethod(BufferResponse = false)]
[SoapDocumentMethod(ParameterStyle = SoapParameterStyle.Bare)]
public FileData DownloadFile(string serverFileName)
{ ... }
```

Самая трудоемкая часть — это реализация специальной сериализации в классе `FileData`. В общем, это выглядит так: создавая объект `FileData` на сервере, мы просто указываем соответствующее имя файла. После сериализации объекта `FileData` и вызова метода `IXmlSerializable.WriteXml()` объект `FileData` создает объект `FileStream` и начинает отправлять двоичные данные, по одному блоку за раз.

Ниже показан базовый скелет класса `FileData`:

```
[XmlRoot(Namespace="http://www.apress.com/ProASP.NET/FileData")]
[XmlSchemaProvider("GetSchemaDocument")]
public class FileData : IXmlSerializable
{
 // Пространство имен для сериализации.
 const string ns = "http://www.apress.com/ProASP.NET/FileData";
 // Путь на стороне сервера.
 private string serverFilePath;
 // При создании объекта FileData необходимо убедиться, что файл существует.
 // Однако от проблем, возникающих при считывании файла (вроде
 // "недостаточно прав", "файл на данный момент используется другим процессом"
 // и т.д.), это не защитит.
 public FileData(string serverFilePath)
 {
   if (!File.Exists(serverFilePath))
   {
       throw new FileNotFoundException("Source file not found.");
   }
   this.serverFilePath = serverFilePath;
 }
 void IXmlSerializable.WriteXml(System.Xml.XmlWriter writer)
 { ... }
 System.Xml.Schema.XmlSchema IXmlSerializable.GetSchema()
 {
   return null;
 }

 void IXmlSerializable.ReadXml(System.Xml.XmlReader reader)
 {
   throw new NotImplementedException();
 }

 public static XmlQualifiedName GetSchemaDocument(XmlSchemaSet xs)
 {
   // Извлекаем путь к файлу схемы.
   string schemaPath = HttpContext.Current.Server.MapPath("FileData.xsd");
   // Извлекаем схему из файла.
   XmlSerializer schemaSerializer = new XmlSerializer(typeof(XmlSchema));
   XmlSchema s = (XmlSchema)schemaSerializer.Deserialize(
       new XmlTextReader(schemaPath), null);
   xs.XmlResolver = new XmlUrlResolver();
   xs.Add(s);
   return new XmlQualifiedName("FileData", ns);
 }
}
```

Нетрудно заметить, что этот класс поддерживает запись данных файла в XML, но не их считывание. Причина проста: перед нами серверная версия кода. Этот код отправляет объекты `FileData`, но не получает их.

В этом примере мы хотим создать XML-представление, в котором данные разделяются на отдельные фрагменты в кодировке Base64. Вот как оно будет выглядеть:

```
<FileData xmlns="http://www.apress.com/ProASP.NET/FileData">
 <fileName>sampleFile.xls</fileName>
 <size>66048</size>
 <content>
 <chunk>...</chunk>
 <chunk>...</chunk>
 ...
 </content>
</FileData>
```

Ниже показана реализация метода `WriteXml()`, выполняющая свою функцию:

```
void IXmlSerializable.WriteXml(System.Xml.XmlWriter writer)
{
 // Открываем файл (в то же время предусматривая
 // возможность его открытия другими потоками).
 FileStream fs = new FileStream(serverFilePath, FileMode.Open,
 FileAccess.Read, FileShare.Read);
 // Записываем имя файла.
 writer.WriteElementString("fileName", ns, Path.GetFileName(serverFilePath));
 // Записываем размер файла (удобно для определения сведений о ходе процесса).
 long length = fs.Length;
 writer.WriteElementString("size", ns, length.ToString());
 // Создаем содержимое файла.
 writer.WriteStartElement("content", ns);
 // Считываем содержимое буфера 4 Кбайт и записываем его
 // фрагментами немного меньшего размера в кодировке Base64).
 int bufferSize = 4096;
 byte[] fileBytes = new byte[bufferSize];
 int readBytes = bufferSize;
 while (readBytes > 0)
 {
   readBytes = fs.Read(fileBytes, 0, bufferSize);
   writer.WriteStartElement("chunk", ns);

   // Этот метод явно кодирует данные. В случае использования
   // другого метода, не исключено попадание в поток XML
   // недействительных символов.
   writer.WriteBase64(fileBytes, 0, readBytes);
   writer.WriteEndElement();
   writer.Flush();
 }
 fs.Close();
 // Завершаем запись XML-данных.
 writer.WriteEndElement();
}
```

Теперь можно завершить код веб-службы. Показанный здесь метод `DownloadFile()` выполняет поиск указанного пользователем файла в жестко закодированном каталоге. Он создает новый объект `FileData` с полным именем пути и возвращает его. Далее в игру вступает код сериализации объекта `FileData`, который считывает файл и начинает записывать его в поток ответа.

```
public class FileService : System.Web.Services.WebService
{
 // Разрешаем загружать файл только в этот каталог.
 string folder = @"c:\Downloads";
 [WebMethod(BufferResponse = false)]
 [SoapDocumentMethod(ParameterStyle = SoapParameterStyle.Bare)]
 public FileData DownloadFile(string serverFileName)
 {
   // Проверяем, чтобы пользователь указал только
   // имя файла (а не весь путь к нему).
   serverFileName = Path.GetFileName(serverFileName);
   // Извлекаем весь путь, используя имя каталога загрузки.
   string serverFilePath = Path.Combine(folder, serverFileName);
   // Возвращаем данные файла.
   return new FileData(serverFilePath);
 }
}
```

Вы можете протестировать этот метод с помощью отображаемой в браузере тестовой страницы и убедиться в том, что данные разбиваются на отдельные фрагменты, посмотрев на XML.

Клиентская сторона

На стороне клиента необходима возможность, позволяющая извлекать данные по одному фрагменту за раз и записывать их в файл. Чтобы предоставить такую возможность, следует изменить прокси-класс так, чтобы он возвращал специальный тип `IXmlSerializable`. В этом классе будет находиться код десериализации.

Совет. Вы можете реализовать код сериализации и код десериализации в одном и том же классе и передать этот класс в виде компонента клиенту и серверу. Однако, как правило, лучше, когда между обоими концами приложения веб-службы сохраняется четкое различие. Это упрощает процедуру обновления клиента до более новых версий.

При создании прокси-класса система .NET попытается создать подходящую копию класса `FileData`. Однако ей это не удастся. Без информации о схеме она будет просто пытаться преобразовать возвращаемое значение в объект `DataSet`. Даже если вы добавите информацию о схеме, все, что .NET сможет сделать — это создать представление класса, в котором все детали (имя, размер и содержимое) будут отображаться в виде отдельных свойств. Такой класс не будет разбивать данные на фрагменты; вместо этого он будет пытаться загрузить в память все сразу.

Чтобы устранить эту проблему, придется вручную настроить прокси-класс. Если речь идет о создании веб-клиента, сначала доведется сгенерировать этот прокси-класс с помощью утилиты командной строки `wsdl.exe`, так чтобы код стал доступным. Ниже показано изменение, которое необходимо внести:

```
public FileDataClient DownloadFile(string serverFileName)
{
 object[] results = this.Invoke("DownloadFile", new object[] {
 serverFileName});
 return ((FileDataClient)(results[0]));
}
```

Очевидно, что внесение изменения в прокси-класс является не очень практичным решением, потому что при каждом обновлении прокси-класса это изменение будет удаляться. Более надежный вариант — реализовать расширение импортера схем, как описывается в следующем разделе.

Вот как выглядит базовая структура класса `FileDataClient`:

```
[XmlRoot(Namespace="http://www.apress.com/ProASP.NET/FileData")]
public class FileDataClient : IXmlSerializable
{
 private string ns = "http://www.apress.com/ProASP.NET/FileData";
 // Место для размещения загружаемого файла.
 private static string clientFolder;
 public static string ClientFolder
 {
   get { return clientFolder; }
   set { clientFolder = value; }
 }
 void IXmlSerializable.ReadXml(System.Xml.XmlReader reader)
 { ... }
 System.Xml.Schema.XmlSchema IXmlSerializable.GetSchema()
 {
   return null;
 }
 void IXmlSerializable.WriteXml(System.Xml.XmlWriter writer)
 {
   throw new NotImplementedException();
 }
}
```

Особое внимание стоит обратить на статическое свойство `ClientFolder`, которое отслеживает место, где должны сохраняться все загружаемые файлы. Значение для этого свойства должно устанавливаться до начала процесса загрузки, потому что метод `ReadXml()` использует эту информацию для того, чтобы определить, где создавать файл. Свойство `ClientFolder` должно быть статическим, поскольку клиенту не предоставляется возможность создавать и конфигурировать объект `FileDataClient`, который он хочет использовать. Вместо этого .NET создает экземпляр объекта `FileDataClient` автоматически и использует его для десериализации данных. Используя статическое свойство, клиент может устанавливать требуемое значение до начала загрузки, как показано ниже:

```
FileDataClient.ClientFolder = @"c:\MyFiles";
```

Код десериализации выполняет задачу, прямо противоположную той, которую выполняет код сериализации: он просматривает фрагменты данных и записывает их в новый файл. Ниже показан этот код полностью:

```
void IXmlSerializable.ReadXml(System.Xml.XmlReader reader)
{
 if (FileDataClient.ClientFolder == "")
 {
   throw new InvalidOperationException("No target folder specified.");
 }
 reader.ReadStartElement();
 // Извлекаем исходное имя файла.
 string fileName = reader.ReadElementString("fileName", ns);
 // Извлекаем размер (не используемый на текущий момент).
 double size = Convert.ToDouble(reader.ReadElementString("size", ns));
 // Создаем файл.
 FileStream fs = new FileStream(Path.Combine(ClientFolder, fileName),
 FileMode.Create, FileAccess.Write);
 // Считываем XML-данные и записываем их в файл, по одному блоку за раз.
 byte[] fileBytes;
 reader.ReadStartElement("content", ns);
 double totalRead = 0;
```

```
while (true)
{
   if (reader.IsStartElement("chunk", ns))
   {
      string bytesBase64 = reader.ReadElementString();
      totalRead += bytesBase64.Length;
      fileBytes = Convert.FromBase64String(bytesBase64);
      fs.Write(fileBytes, 0, fileBytes.Length);
      fs.Flush();
      // Здесь можно было бы сообщить о ходе процесса
      // путем вызова соответствующего события.
      Console.WriteLine("Received chunk.");
   }
   else
   {
      break;
   }
}
fs.Close();
reader.ReadEndElement();
reader.ReadEndElement();
}
```

Ниже показан код консольного приложения, которое использует веб-службу `FileService`:

```
static void Main()
{
   Console.WriteLine("Downloading to c:\\");
   FileDataClient.ClientFolder = @"c:\";
   Console.WriteLine("Enter the name of the file to download.");
   Console.WriteLine("This is a file in the server's download directory.");
   Console.WriteLine("The download directory is c:\\temp by default.");
   Console.Write("> ");
   string file = Console.ReadLine();
   FileService proxy = new FileService();
   Console.WriteLine();
   Console.WriteLine("Starting download.");
   proxy.DownloadFile(file);
   Console.WriteLine("Download complete.");
}
```

Результаты выполнения этого кода можно видеть на рис. 36.8.

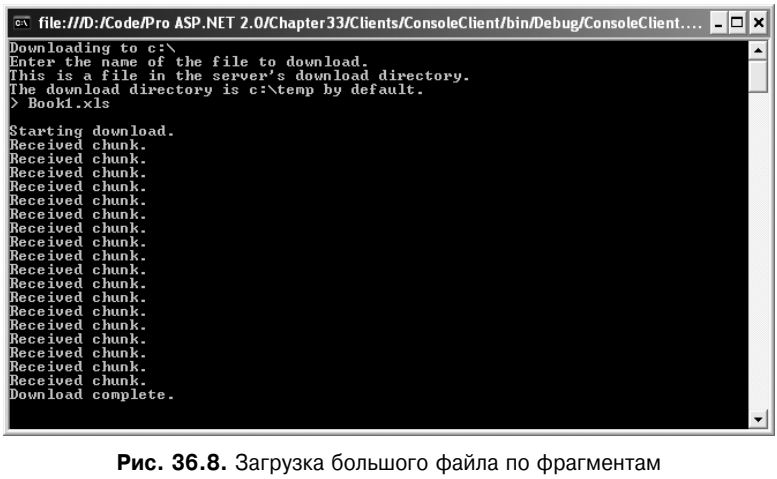

Рис. 36.8. Загрузка большого файла по фрагментам

Этот пример можно найти и на веб-сайте издательства, с несколькими незначительными изменениями (вроде того, что клиентские и серверные методы для работы с файлом там объединены в один класс `FileData`).

Расширения импортера схем

Одним из ключевых принципов проектирования, ориентированного на службы, является то, что клиент и сервер совместно используют контракты, а не классы. Такой уровень абстракции позволяет клиентам, функционирующим на самых разных платформах, взаимодействовать с одной и той же веб-службой. Они отправляют одинаковые сериализуемые XML-данные, но имеют возможность использовать разные программные структуры (такие как классы) для подготовки своих сообщений.

В некоторых случаях вы можете захотеть нарушить эти правила, чтобы позволить своим клиентам работать с многофункциональными типами данных. Например, вы могли бы поступить следующим образом: взять какой-нибудь специальный класс данных, передать его и клиенту, и серверу, и затем позволить им отправлять или получать экземпляры этого класса с помощью веб-службы. .NET 2.0 делает это вполне возможным с помощью новой опции, которая называется расширениями импортера схем (schema importer extensions).

Совет. Дважды подумайте, прежде чем использовать сложные типы данных. Такой подход опасен тем, что он запросто может привести к созданию очень специализированной (патентованной) веб-службы. Хотя ваша веб-служба и будет использовать XML-данные (которые всегда могли считываться на любой платформе), как только вы начнете подгонять эти XML-данные под специфические типы платформы, они могут стать настолько сложными, что другие клиенты окажутся просто не способными выполнять их анализ или делать с ними что-либо полезное. Например, большинство клиентов, отличных от .NET, не имеет возможности обрабатывать XML-данные, генерируемые для типа `DataSet`.

Прежде чем создавать расширение импортера схем, удостоверьтесь в наличии в коде веб-службы атрибута `XmlSchemaProvider`, используемого для обозначения метода, который возвращает информацию о схеме. Без информации о схеме у средства генерации прокси-класса не будет информации, необходимой ему для определения специальных типов данных, поэтому любые создаваемые импортеры схем окажутся просто бесполезными.

В случае класса `FileData` схема извлекается из такого файла схемы:

```
<xs:schemaid="FileData"targetNamespace=http://www.apress.com/ProASP.NET/FileData
   elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:complexType name="FileData" >
  <xs:sequence>
   <xs:element name="fileName" type="xs:string" />
   <xs:element name="size" type="xs:int" />
   <xs:element name="content" >
    <xs:complexType >
     <xs:sequence>
       <xs:element name="chunk" type="xs:base64Binary" maxOccurs="unbounded"/>
     </xs:sequence>
    </xs:complexType>
   </xs:element>
  </xs:sequence>
 </xs:complexType>
</xs:schema>
```

Теперь можно приступать к разработке импортера схем, позволяющего клиенту распознавать этот тип данных.

Чтобы использовать импортер схем, необходимо выполнить два действия — создать расширение и затем зарегистрировать его. Чтобы создать расширение, создайте новый компонент библиотеки классов (т.е. DLL сборки). В этой сборке добавьте класс, породив его от класса `SchemaImporterExtension`.

Когда генератор прокси-класса наталкивается на сложный тип (когда он генерирует прокси-класс), он вызывает метод `ImportSchemaType()` каждого расширения импортера схем, которое определено в файле `machine.config`. Каждый импортер схемы может проверить пространство имен и схему типа и затем решить обработать его за счет отображения XML-типа на тип, известный .NET.

Ниже показан пример с импортером `FileDataSchemaImporter`, который конфигурирует прокси-класс так, чтобы он мог использовать класс `FileDataClient`:

```
public class FileDataSchemaImporter : SchemaImporterExtension
{
 public override string ImportSchemaType(string name, string ns,
   XmlSchemaObject context, XmlSchemas schemas, XmlSchemaImporter importer,
   CodeCompileUnit compileUnit, CodeNamespace mainNamespace,
   CodeGenerationOptions options, CodeDomProvider codeProvider)
 {
  if (name.Equals("FileData") &&
    ns.Equals("http://www.apress.com/ProASP.NET/FileData"))
  {
       mainNamespace.Imports.Add(new CodeNamespaceImport("FileDataComponent"));
       return "FileDataClient";
  }
  else
  {
       // Выбираем не обрабатывать тип.
       return null;
  }
 }
}
```

Это очень простой импортер схемы. Он делает две вещи.

- Указывает прокси-классу использовать для этого типа класс `FileDataClient`. Это означает, что прокси-класс будет использовать существующий класс, и не будет автоматически генерировать клиентскую версию класса `FileData` (стандартное поведение).
- Указывает генератору прокси-класса добавить импорт пространства имен для пространства имен `FileDataComponent`. Однако обязательно необходимо убедиться в том, что сборка с классом `FileDataComponent.FileData` доступна в проекте.

Создав импортер схемы, вы должны установить его в глобальном кэше сборок. Присвойте ему строгое имя (на вкладке Signing (Подпись) в окне свойств проекта) и затем перетащите его в каталог `c:\[КаталогWindows]\Assembly` или воспользуйтесь утилитой командной строки `gacutil.exe`.

После того как импортер схемы будет безопасно установлен в кэше, с помощью проводника Windows отыщите его маркер открытого ключа. Вооружившись этой информацией, вы можете зарегистрировать свой импортер схемы в файле `machine.config`, используя следующие параметры настройки:

```
<configuration>
...
 <system.xml.serialization>
  <schemaImporterExtensions>
   <add name="FileDataSchemaImporter" type=
    "SchemaImporter.FileDataSchemaImporter, SchemaImporter, Version=1.0.0.0,
    Culture=neutral, PublicKeyToken=6c8e0bfd71c11c40" />
  </schemaImporterExtensions>
 </system.xml.serialization>
</configuration>
```

Атрибут типа — это очень важная часть. Формат должен быть следующим (все это одна строка):

```
<Имя класса, включающее имя пространства имен>, <Имя сборки без расширения>,
<Версия>, <Региональный стандарт>, <Маркер открытого ключа>
```

Теперь импортер схем готов к использованию. Попробуйте запустить утилиту `wsdl.exe` для службы `FileService`:

```
http://localhost/WebServices2/FileService.asmx
```

Сгенерированный прокси-класс будет включать имя типа, который вы указали, и импорт нового пространства имен. Однако класс `FileData` создан не будет — взамен вы увидите специальную версию, которую создали в компоненте `FileData`.

Добавьте этот сгенерированный прокси-класс в проект клиента. Теперь вы сможете загружать файлы с поддержкой пофрагментной передачи данных, которую обеспечивает класс `FileData`.

На заметку! На текущий момент импортеры схем использует только утилита `wsdl.exe`. Они не используются при создании веб-ссылки с помощью Visual Studio. Это поведение планируется изменить в следующих версиях.

Резюме

В этой главе мы подробно рассмотрели два наиболее важных протокола веб-служб — SOAP и WSDL. SOAP — это чрезвычайно простой протокол для обмена сообщениями. WSDL — это гибкий, расширяемый протокол для описания веб-служб. Вместе они гарантируют, что в ближайшие годы веб-службы можно будет создавать и использовать практически на любой платформе программирования. Также в этой главе было показано, как настраивать XML-данные, возвращаемые веб-службой.