

KAPITOLA 33

Standardy webové služby a rozšíření

V předchozí kapitole jste se naučili, jak .NET skrývá nízkourovňový podklad webových služeb, čímž vám umožňuje vytvářet a používat sofistikované webové služby, aniž byste museli znát podrobnosti o nižších úrovních protokolů, které používáte. Tato abstrakce vyšší úrovně je hlavním tématem moderního programování. Vývojáři Windows si například dělají starosti s jednotlivými pixely svých aplikací; vývojáři ASP.NET jenom zřídka potřebují napsat do výstupního datového proudu samotné značky. Samozřejmě – pracovní rámec vyšší úrovně někdy nemusí stačit. Například vývojáři Windows, kteří vytvářejí hry v reálném čase, občas potřebují pracovat s hardwarem nižší úrovně; weboví vývojáři, vytvářející vlastní ovládací prvky se pravděpodobně budou muset ponořit do pichlavé změti JavaScriptu a HTML.

Stejný princip také platí u webových služeb. Ve většině případů potřebujete v .NET pouze model webových služeb vyšší úrovně. Ten vám zajistí vám rychlé, produktivní programování, odolné vůči chybám. Občas se však potřebujete ponořit do detailů. Platí to zejména tehdy, potřebujete-li zasílat komplexní objekty klientům, kteří nepoužívají .NET, nebo vytvořit rozšíření, které se zapojí do modelu webových služeb .NET. V této kapitole se společně podíváme právě na tuto nižší úroveň a naučíme se více o výchozích protokolech SOAP a WSDL.

TIP Pokud jste odborníkem na webové služby a zajímáte se o nové funkce .NET 2.0, pozorně si prostudujte sekci "Přizpůsobení zpráv SOAP" na konci této kapitoly. Tato sekce popisuje, jak řídit proces serializace pomocí `IXmlSerializable` a jak vytvářet rozšíření importéru schématu, které vám umožní používat typy, které webové služby běžně nepodporují. Vyhledejte si také sekci "Implementace existujícího kontraktu".

Součinnost WS

Webové služby se vyvinuly velice rychle, přičemž standardy jako SOAP a WSDL se pořád ještě vyvíjejí. V nejstarších sadách nástrojů webových služeb jednotliví výrobci interpretovali části těchto standardů poněkud rozdílně, což mělo za následek problémy se součinností. Některé funkce původních standardů jsou však již dnes považovány za zastaralé.

Řešení těchto malých rozdílů představuje takové malé minové pole, zejména tehdy, pokud potřebujete vytvořit webové služby, ke kterým budou přistupovat klienti používající jiné programovací platformy a sady

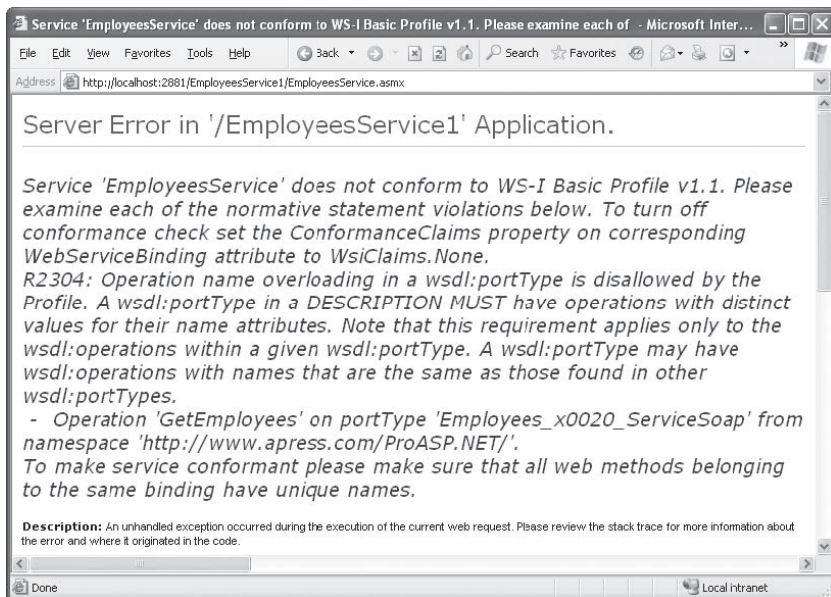
nástrojů webových služeb. Naštěstí se v poslední době objevil další standard, který specifikuje širokou škálu pravidel a doporučení, a který je navržen tak, aby zaručoval součinnost jednotlivých implementací webových služeb od různých výrobců. Tímto dokumentem je WS-Interoperability Basic Profile (viz <http://www.ws-i.org>). Specifikuje doporučenou podmnožinu specifikací SOAP 1.1 a WSDL 1.1 a stanovuje několik základních pravidel. Tento dokument je silně podporován všemi výrobci webových služeb (včetně Microsoftu, IBM, Sun a Oracle).

Za ideálních podmínek byste si jako vývojáři neměli dělat vrásky se specifikacemi uvedenými v tomto dokumentu, protože .NET by mělo pravidla v něm uvedená standardně respektovat. Nová verze .NET 2.0 tento problém řeší prostřednictvím atributu `WebServiceBinding`, který je automaticky přidáván do vaší třídy webové služby, jakmile ji vytvoříte ve Visual Studiu:

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

Atribut `WebServiceBinding` označuje úroveň součinnosti, které chcete dosáhnout. V současnosti je jedinou volbou `WsiProfiles.BasicProfile1_1`, která reprezentuje dokument WS-Interoperability Basic Profile 1.1. S dalším vývojem standardů se však určitě budou objevovat novější verze SOAP a WSDL, a s nimi také novější verze tohoto standardizačního dokumentu.

Jakmile je atribut `WebServiceBinding` na svém místě, .NET vám ohlásí závažnou chybu při kompilaci, pokud se chování vaší webové služby dostane mimo povolené meze. Webové služby .NET jsou standardně vyhovující, nicméně neúmyslně můžete přidáním určitých atributů vytvořit nevyhovující službu. Například je možné vytvořit dvě webové metody se stejným názvem, pokud mají odlišné signatury, a pokud jim přiřadíte odlišné alternativní názvy pomocí vlastnosti `MessageName` atributu `WebMethod`. Podle standardizačního dokumentu toto chování ovšem není povoleno a pokusíte-li se spustit takovou webovou službu, bude vygenerována chybová stránka, kterou vidíte na obrázku 33-1.



Obrázek 33-1. SOAP zpráva.

Shodu s vlastností `EmitConformanceClaims` můžete indikovat následujícím způsobem:

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

V tom případě jsou do WSDL dokumentu vloženy další informace, které indikují, že webová služba je ve shodě se standardem. Je důležité pochopit, že je to pouze pro informaci – vaše webová služba totiž může být ve shodě, aniž byste to museli explicitně prohlašovat.

V ojedinělých případech se můžete rozhodnout porušit jedno z pravidel součinnosti WS, abyste vytvořili webovou službu, kterou budou moci používat i starší nevyhovující aplikace. V této situaci musíte nejdříve vypnout shodu (compliance) odstraněním atributu `WebServiceBinding`. Další způsob je vypnout ověřování shody a tuto skutečnost zdokumentovat pomocí atributu `WebServiceBinding` bez profilu:

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

SOAP

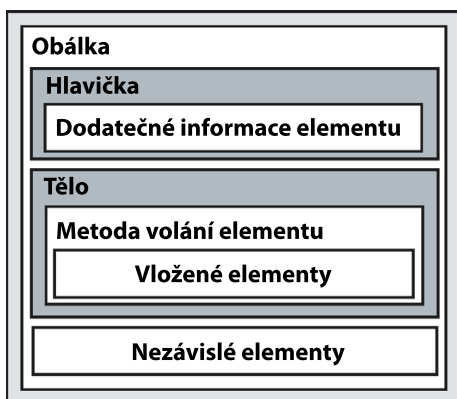
SOAP je meziplatformový standard, který je používán pro formátování zpráv, zasílaných mezi webovými službami a klientskými aplikacemi. Krása standardu SOAP spočívá v jeho flexibilitě. SOAP můžete použít nejenom pro zasílání jakýchkoliv typů XML dat (včetně vašich proprietárních XML dokumentů), ale můžete jej použít také s jinými transportními protokoly, než je klasické HTTP. SOAP zprávy můžete například zasílat přes přímé TCP/IP spojení.

POZNÁMKA .NET Framework obsahuje podporu pouze pro nejběžnější způsobu použití SOAP, a sice zasílání SOAP přes HTTP. Pokud potřebujete flexibilnější přístup, měli byste zvážit použití sady nástrojů WSE (Web Services Enhancements) od Microsoftu, více informací naleznete v další kapitole.

SOAP je poměrně jednoduchý standard. Jeho prvním principem je, že SOAP zpráva je dokumentem XML. Tento XML dokument má jediný kořenový prvek `<Envelope>`, který slouží jako SOAP obálka. Zbytek zprávy je uložen uvnitř obálky, která obsahuje záhlaví a tělo.

Webové služby .NET v zásadě používají dva typy SOAP zpráv. Klient zasílá webové službě zprávu o požadavku, čímž spustí webovou metodu. Po ukončení zpracování webová služba zasílá zpět zprávu odpovědi. Obě tyto zprávy mají stejný formát, který je zobrazen na obrázku 33-2.

POZNÁMKA Slovo SOAP bylo původně zkratkou slov Simple Object Access Protocol. Pro současné verze SOAP standardu to ovšem již neplatí. Podrobné informace o SOAP specifikacích naleznete na <http://www.w3.org/TR/soap>.



Obrázek 33-2. SOAP zpráva.

Kódování SOAP

Existují dva odlišné (avšak úzce spojené) styly SOAP. První je tzv. styl dokumentu SOAP zobrazuje data, která jsou vyměňována jako dokumenty. Řečeno jinými slovy – každá zasílaná či přijímaná SOAP zpráva obsahuje v těle XML dokument. Druhý je tzv. RPC styl SOAP, který zobrazuje výměnu dat jako volání metody u vzdálených objektů. Vzdáleným objektem může být objekt Javy, COM komponenta, objekt .NET nebo něco úplně jiného. V RPC stylu SOAP je nejvzdálenější element v požadavku vždy pojmenován podle metody, přičemž každý parametr dané metody má svůj element. U odpovědi má nejvzdálenější element stejný název jako metoda s přidaným textem Response.

Když nyní vidíte, jakým způsobem pojalý webové služby .NET objektově orientovaný RPC model, mohli byste se domnívat, že tyto služby používají RPC styl SOAP. Není to však pravda. Důvod je jednoduchý – styl dokumentu SOAP (document-style SOAP) je mnohem flexibilnější a umožňuje vyměňovat libovolné XML dokumenty mezi webovou službou a zákazníkem této služby. Přestože .NET používá styl dokumentu SOAP, své zprávy formátuje podobně jako RPC styl SOAP a používá přitom mnoho stejných konvencí.

A aby byl život ještě zajímavější, SOAP data mohou být zakódována dvěma způsoby – jako literal a SOAP sekce 5 (SOAP section 5). Kódování literal značí, že data jsou zakódována jako specifické XML schéma. Kódování SOAP sekce 5 značí, že data jsou zakódována podle podobných, avšak přísnějších pravidel, stanovených specifikací sekce 5 (section 5) standardu SOAP. Pravidla sekce 5 představují tak trochu krok zpět. Důvod, proč tato pravidla existují, je ten, že SOAP bylo vyvinuto před dokončením standardu Schéma XML.

POZNÁMKA Všechny webové služby .NET standardně používají styl dokumentu SOAP s kódováním literal. Toto chování byste měli změnit pouze v případě, že potřebujete dosáhnout kompatibility s nějakou odvozenou aplikací.

V této chvíli se možná zeptáte, proč potřebujete znát tyto podrobnosti o nižší úrovni SOAP. Ve většině případů je znát nepotřebujete. Mohlo by se však stát, že budete chtít změnit celkové kódování vaší webové služby. Jeden z důvodů by mohl být například ten, že budete chtít vystavit webovou metodu, která může být vyvolána klientem, který podporuje pouze RPC styl SOAP. Ačkoliv je tento scénář čím stálo vzácnější (navíc porušuje WS-Interoperability Basic Profile), může se přesto vyskytnout.

ASP.NET obsahuje dva atributy (oba se nacházejí ve jmenném prostoru System.Web.Services.Protocols), které můžete použít k řízení celkového kódování všech metod webové služby:

- **SoapDocumentService.** Tento atribut použijete, pokud chcete, aby všechny metody používaly styl dokumentu SOAP (který je stejně standardní). Pomocí parametru SoapBindingUse však můžete specifikovat kódování SOAP sekce 5 místo kódování document/literal.
- **SoapRpcService.** Tento atribut použijte, pokud chcete, aby všechny webové služby používaly RCP styl SOAP s kódováním SOAP sekce 5.

U jednotlivých webových metod můžete použít následující atributy:

- **SoapDocumentMethod.** Tento atribut přidejte, pokud chcete použít styl dokumentu SOAP pro jedinou webovou metodu. Můžete specifikovat kódování, které má být použito.
- **SoapRpcMethod.** Tento atribut přidejte, pokud chcete použít RPC styl SOAP pro jedinou webovou metodu.

Toto je užitečné, pokud chcete zpřístupnit ve webové službě dvě metody, které sice vykonávají podobnou funkci, nicméně podporují odlišný typ SOAP kódování. V této kapitole se však soustředíme na styly SOAP zpráv, který .NET standardně používá (document/literal).

Verze SOAP

V dnešní době je nejpoužívanější verzí SOAP verze SOAP 1.1. Jedinou další variantou je novější SOAP 1.2, který objasňuje mnoho aspektů SOAP standardu, zavádí několik menších vylepšení a formalizuje model rozšiřitelnosti.

POZNÁMKA Bez ohledu na to, jakou verzi SOAP používáte, jsou schopnosti webové služby .NET stejné. V podstatě – jediným důvodem, proč byste se měli zajímat o to, jaká verze se používá, je zajištění kompatibility s klienty, kteří nepoužívají .NET. V příkladech této kapitoly je použit SOAP 1.1.

.NET verze 1.x podporovalo pouze SOAP 1.1. Webová služba, která je vytvořena v .NET 2.0 však automaticky podporuje jak SOAP 1.1, tak i SOAP 1.2. Pokud to chcete změnit, prostřednictvím souboru web.config můžete zakázat jeden z těchto standardů:

```
<configuration>
  <system.web>
    <webServices>
      <protocols>
        <!-- Toto použijte pro vypnutí SOAP 1.2 -->
        <remove name="HttpSoap12"/>

        <!-- Toto použijte pro vypnutí SOAP 1.1 -->
        <remove name="HttpSoap"/>
      </protocols>
    </webServices>
    ...
  </system.web>
</configuration>
```

Když v .NET vytvoříte třídu proxy, standardně používá SOAP 1.1, pokud ovšem není dostupný SOAP 1.2. Toto chování můžete ovšem programově překrýt nastavením vlastnosti `SoapVersion` třídy proxy předtím, než zavoláte jakoukoliv webovou metodu:

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

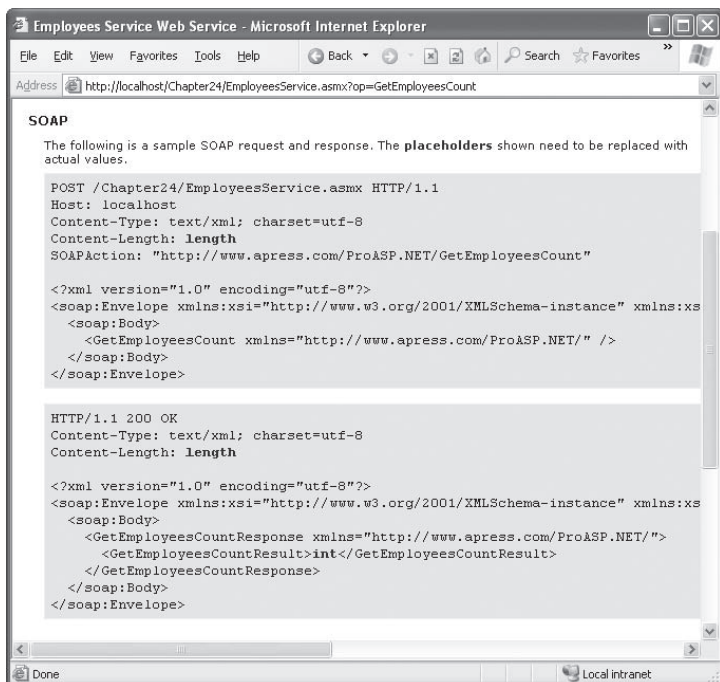
Další způsob je vytvořit třídu proxy, která vždy standardně používá SOAP 1.2, a to pomocí přepínače příkazového řádku `/protocol:SOAP12` utility `wsdl.exe`.

Trasování SOAP zpráv

Než se podrobněji podíváme na standard SOAP, stojí za to si popsat, jakým způsobem se můžete dívat na SOAP zprávy zasílané webové službě .NET a zpět. Bohužel .NET neobsahuje žádné nástroje pro trasování a ladění SOAP zpráv. Poměrně snadno se však můžete podívat na odpovídající zprávy SOAP pomocí jiných nástrojů.

Prvním přístupem je použití testovací stránky prohlížeče. Jak víte, tato stránka nepoužívá SOAP, nýbrž zredukováný protokol HTTP POST, který kóduje data jako dvojice jméno/hodnota. Testovací stránka však neobsahuje příklad, jak by měla vypadat SOAP zpráva pro jednotlivé webové metody.

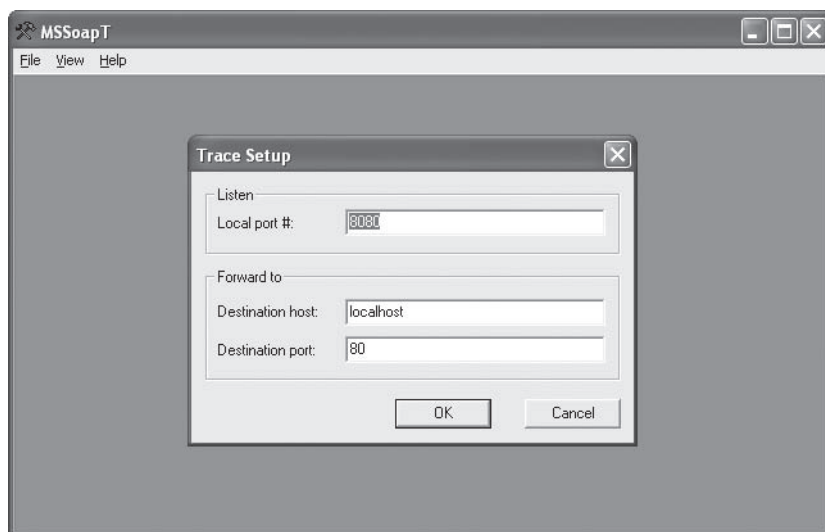
Vezměte si například webovou službu `EmployeesService`, která byla vytvořena v předchozí kapitole. Když nahrajete testovací stránku, kliknete na odkaz směřující na metodu `GetEmployeesCount()` a odrolujete stránkou níže, uvidíte vzorovou zprávu o požadavku a zprávu o odpovědi s vyhrazenými místy, kam by se měly přijít hodnoty dat. Obrázek 33-3 ilustruje část těchto dat. Můžete také odrolovat stránkou ještě níže, abyste viděli formát jednodušších HTTP POST zpráv.



Obrázek 33-3. Vzorové SOAP zprávy pro metodu `GetEmployeesCount()`.

Tyto příklady vám sice pomohou pochopit standard SOAP, nicméně vám neposlouží, pokud budete chtít vidět skutečné SOAP zprávy – například tehdy, pokud budete chtít vyřešit neočekávaný problém týkající se kompatibility webové služby .NET s nějakým jiným klientem. Naštěstí existuje jednoduchý způsob, jak zachytit skutečné SOAP zprávy během jejich přenosu přes síť, musíte však použít jiný nástroj. Jedná se o Microsoft SOAP Toolkit, což je COM knihovna, která obsahuje objekty, které vám umožní používat webové služby v jazycích založených na COM, jako je Visual Basic 6 a Visual C++ 6. Kromě těchto nástrojů obsahuje SOAP Toolkit i nepostradatelný trasovací nástroj, pomocí kterého nahlížíte pod pokličku SOAP komunikace.

SOAP Toolkit si můžete stáhnout na adrese: http://msdn.microsoft.com/webservices/_building/soaptk. Jakmile máte SOAP Toolkit nainstalovaný, můžete spustit trasovací utilitu tak, že z menu Start vyberete položku Microsoft SOAP Toolkit > Trace Utility. Po zavedení trasovací utility vyberte volbu File > New > Formatted Trace. Objeví se vám okno, které vidíte na obrázku 33-4.



Obrázek 33-4. Spuštění nového SOAP trasování.

Výchozí nastavení indikují, že trasovací utilita bude naslouchat komunikaci na portu 8080, přičemž všechny zprávy bude přeposílat portu 80 (na kterém IIS naslouchá nezašifrované HTTP komunikaci, včetně požadavků GET a POST a SOAP zpráv). Klikněte na OK, čímž tato nastavení potvrdíte.

Nyní potřebujete ještě jednu věc. Klienti vaší webové služby budou standardně obcházet trasovací nástroj tím, že své SOAP zprávy budou zasílat přímo na port 80, a nikoliv na port 8080. Kód klienta musíte nastavit tak, aby SOAP zprávy posílal na port 8080. To provedete jednoduše pomocí URL adresy, ve které specifikuje port, jak vidíte zde:

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

Chcete-li změnit URL, musíte modifikovat vlastnost `Url` třídy proxy ještě předtím, než zavoláte jakoukoliv z jeho metod. URL adresu ovšem nemusíte natvrdo zakódovat – můžete použít zde uvedený kód, který pomocí třídy `System.Uri` obecně přeměňuje jakoukoliv URL na port 8080:

```
// Vytvoř proxy.  
EmployeesService proxy = new EmployeesService();  
Uri newUrl = new Uri(proxy.Url);
```



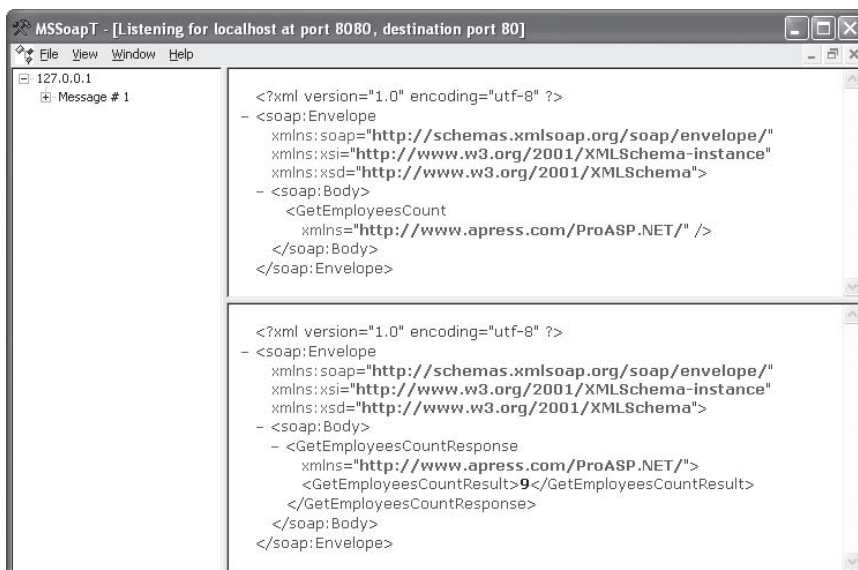
```

proxy.Url = newUrl.Scheme + "://" + newUrl.Host + ":8080"
+ newUrl.AbsolutePath;
// Vyvolej webovou službu a získej výsledky.
DataSet ds = proxy.GetEmployeesCount();

```

Podobnou změnu nemusíte provádět ve vaší webové službě, protože toto automaticky odešle svoji zprávu odpovědi zpět na port, na kterém vznikla zpráva požadavku, v našem případě tedy na port 8080. Trasovací utilita následně zaznamená zprávu odpovědi a předá ji zpět klientské aplikaci.

Jakmile jste ukončili volání webové služby, můžete rozšířit strom trasovací utility tak, abyste se mohli podívat na zprávy požadavku a odpovědi. Na obrázku 33-5 je zobrazen výsledek po spuštění předchozího fragmentu kódu. V horním okně vidíte u metody `GetEmployeesCount()` zprávu o požadavku. V dolním okně je odpověď s aktuálním počtem zaměstnanců nacházejících se v tabulce (tj. devět). Pokud zavoláte více webových metod, do stromu budou přidány další uzly.



Obrázek 33-5. Zachytávání SOAP zpráv.

Trasovací nástroj SOAP je vhodný pro prohlížení SOAP zpráv, zejména tehdy, pokud chcete vidět, jak je serializován neobvyklý datový typ, otestovat vlastní rozšíření, nebo pokud chcete řešit problémy spojené se součinností. Nejlepší na tom je, že na webový server nemusíte instalovat žádný speciální software. Místo toho jednoduše předáte klientské zprávy lokální trasovací utilitě.

V následujících sekcích se blíže podíváme na formát SOAP. Je možné, že budete chtít použít trasovací utilitu SOAP pro otestování těchto příkladů a možná se budete chtít podívat na výchozí SOAP zprávy.

SOAP obálka

Každá SOAP zpráva je uzavřena do kořenového elementu `<Envelope>`. Uvnitř obálky se nachází nepovinný element `<Header>` a povinný element `<Body>`. Zde uvádíme její základní kostru:

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



```
<soap:Header>
</soap:Header>
<soap:Body>
</soap:Body>
</soap:Envelope>
```

Všimněte si, že všechny elementy <Envelope>, <Body> a <Header> existují ve jmenném prostoru obálky SOAP. Tato část je povinná.

Element <Body> obsahuje vlastní tělo zprávy. Zde se umísťují aktuální data, jako třeba parametry zprávy požadavku, nebo návratovou hodnotu zprávy odpovědi. Je rovněž možné specifikovat chybovou informaci, která označí chybový stav, a nezávislé elementy, které definují serializaci komplexních typů.

Zprávy o požadavku

U automaticky generovaných webových služeb .NET označuje první element <Body> název metody, kterou voláte. Zde je uvedena kompletní SOAP zpráva pro zavolání metody `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>
```

V tomto případě potřebujete pouze prázdný element <GetEmployeesCount>. Pokud však metoda vyžaduje jakékoliv informace (v podobě parametrů), budou tyto informace zakódovány v elementu <GetEmployeesCount> s odpovídajícím názvem parametru. Následující SOAP například reprezentuje zavolání metody `GetEmployeesByCity()`, která specifikuje město Londýn:

```
<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>
```

Všimněte si, že v obou příkladech mají data uvnitř elementu <Body> přiřazený jmenný prostor webové služby (v našem případě `http://www.apress.com/ProASP.NET`). Protokol SOAP je dostatečně flexibilní na to, aby umožnil do elementu <Body> vložit jakékoliv XML značky. To znamená, že SOAP není brán jako protokol pro vzdálené zavolání metody, ale můžete jej použít pro výměnu komplexních XML dokumentů. V obchodním scénáři mohou být jednotlivé části tohoto dokumentu v automatizovaném pracovním procesu vytvořeny odlišnými společnostmi, a dokonce mohou obsahovat digitální podpisy. Bohužel – programovací model .NET takové postupy komplikuje, protože zaobaluje údaje SOAP a WSDL prostřednictvím objektově orientované abstrakce. Přední vývojáři webových služeb však tyto komplikace již řeší a je pravděpodobné, že příští verze ASP.NET budou mnohem flexibilnější.

POZNÁMKA

Nyní se možná ptáte, zdali je možné vytvořit nekompatibilní SOAP zprávu. Webové služby .NET například používají název metody jako první element v <Body>, jiné implementace webových služeb se však této konvence nemusí držet. Řešením těchto problémů je WSDL standard, kterému se budeme v této kapitole věnovat v sekci WSDL. Tento standard umožňuje, aby .NET mohlo velice podrobně definovat formát zpráv vašich webových služeb. Řečeno jinými slovy – nevadí, pokud se pojmenovávání a organizace SOAP zprávy v .NET trochu odlišuje od konkurenční platformy, protože ostatní platformy naleznou ve WSDL dokumentu .NET příslušná pravidla, která stanoví, jak mají reagovat na webovou službu.

Zprávy o odpovědi

Po odeslání zprávy požadavku klient čeká na odezvu z webového serveru (podobně jako u HTTP požadavku). Tato zpráva odpovědi používá podobný formát jako zpráva požadavku. Na základě konvence .NET (nejedná se o požadavek na specifikaci SOAP), má první dceřiný element v elementu <Body> název metody, k němuž je připojena přípona Response. Například po zavolání metody GetEmployeesCount() můžete obdržet tuto odpověď:

```
<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>
```

Podobně jako zpráva požadavku má i zpráva odpovědi oprávnění používat jmenný prostor webové služby. Uvnitř elementu <GetEmployeesCountResponse> je element <GetEmployeesCountResult> s návratovou hodnotou. Na základě konvence .NET je to název webové metody, k němuž je připojena přípona Result. Zajímavé je, že to je jediná informace, kterou můžete najít uvnitř elementu <XxxResponse>. Pokud metoda používá parametry ref nebo out, budou uvnitř obsaženy také data těchto parametrů. To klientovi umožní aktualizovat hodnoty parametru po ukončení volání, což má za následek stejné chování jako při použití parametrů out či ref při volání lokální metody.

Chybové zprávy

Standard SOAP také definuje způsob označení chybových stavů. Pokud se na serveru vyskytne chyba, je zaslána zpráva a prvním elementem uvnitř elementu <Body> je element <Fault>. Naštěstí .NET tento standard automaticky dodržuje a používá. Pokud se vyskytne neošetřená výjimka během vykonávání webové metody, .NET zašle klientovi chybovou zprávu SOAP. Když třída proxy obdrží chybovou zprávu, vznikne výjimka na straně klienta, která informuje aplikaci klienta. Jak ale uvidíte dále, proces konverze výjimky webové služby na výjimku aplikace klienta není úplně bezproblémový.

Představte si například, co se stane, zavoláte-li metodu GetEmployeesCount() v době, kdy databázový server není k dispozici. Na straně webového serveru vznikne výjimka SqlException, která je zachycena ASP.NET, který vrací následující – poněkud zkrácenou – chybovou zprávu:

```
<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>

```

Element Fault obecně obsahuje elementy <faultcode>, <faultstring> a <detail>. Elementu <faultcode> je přiřazena jedna z těchto předdefinovaných hodnot – ClientFaultCode (vyskytl se problém s klientským požadavkem SOAP), MustUnderstandFaultCode (požadovaná část SOAP zprávy nebyla rozpoznána), ServerFaultCode (na serveru se vyskytla chyba) a VersionMismatchFaultCode (byl nalezen neplatný jmenný prostor). Element <faultstring> pak obsahuje úplný popis problému. Nepovinný element <detail> můžete použít k uložení doplňkových informací o chybě, která se vyskytla (ačkoli v našem příkladu je tento element prázdný).

Problém je v tom, že element <Fault> není mapován přímo do třídy výjimek .NET. Když proxy obdrží tuto zprávu, nedokáže identifikovat objekt původní výjimky a nedokáže ani zjistit, zda je třída výjimky u klienta vůbec k dispozici. Výsledkem je, že třída proxy jednoduše zavolá obecnou SoapException s úplnými údaji ve formě elementu <faultstring>.

Abyste pochopili, jak to funguje, zamyslete se nad tím, co se stane, napíšete-li u vašeho klienta následující kód:

```

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

```

V tomto případě nebude výjimka nikdy zachycena, protože se jedná o SoapException, nikoliv o SqlException (ačkoli základní příčinou problému a současně původním objektem výjimky je SqlException). I když zachytíte SqlException ve webové metodě a ručně zavoláte odlišný objekt výjimky, stejně bude u klienta překonvertován na SoapException. Pro klienta je proto dost obtížné odlišovat jednotlivé typy chybových stavů. Klient dokáže zachytit pouze System.Net.WebException (která představuje uplynutí časové prodlevy nebo obecný problém sítě) nebo System.Web.Services.Protocols.SoapException (která představuje jakoukoliv výjimku .NET, která se vyskytla ve webové službě).

Máte ovšem ještě jednu možnost. Výjimku webové metody můžete zachytit na straně serveru, a pak můžete sami zavolat podporovanou SoapException. Výhoda tohoto přístupu spočívá v tom, že než vaše webová služba zavolá objekt SoapException, můžete jej nakonfigurovat vložení dodatečného XML do elementu <detail>. Klient si poté může přečíst obsah této značky a programově rozhodnout, co se ve skutečnosti stalo.

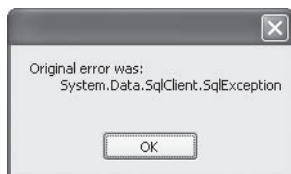
Zde například uvádíme chybovou verzi metody `GetEmployeesCount()`, která používá tento přístup pro přidání názvu typu původní výjimky k `SoapException` pomocí vlastního elementu `<ExceptionType>`. Tento přístup můžete rozšířit pro přidání jakékoliv kombinace elementů, atributů a dat.

```
[WebMethod()]
public int GetEmployeesCountError()
{
    SqlConnection con = null;
    try
    {
        con = new SqlConnection(connectionString);
        // Úmyslně vytvoř chybový SQL řetězec.
        string sql = "INVALID_SQL COUNT(*) FROM Employees";
        SqlCommand cmd = new SqlCommand(sql, con);
        con.Open();
        return (int)cmd.ExecuteScalar();
    }
    catch (Exception err)
    {
        // Vytvoř podrobné informace
        // element <ExceptionType> s názvem typu.
        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);
        // Vytvoř vlastní SoapException.
        // Použij zprávu původní výjimky
        // a přidej podrobné informace.
        SoapException soapErr = new SoapException(err.Message,
            SoapException.ServerFaultCode, Context.Request.Url.AbsoluteUri, node);
        // Vyvolej revidovanou SoapException.
        throw soapErr;
    }
    finally
    {
        con.Close();
    }
}
```

Klientská aplikace přečte element `<ExceptionType>`, aby získala dodatečné informace, které jste přidali. Zde uvádíme příklad, který zobrazuje název výjimky v okně Windows se zprávou (viz obrázek 33-6):

```
EmployeesService proxy = new EmployeesService();
```

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



Obrázek 33-6. Získávání doplňkových chybových informací SOAP.

Záhlaví SOAP

SOAP rovněž specifikuje sekci <Header>, kde můžete umístit důležité informace. V typickém případě se jedná o informace, které nepatří do vlastního těla zprávy. Záhlaví SOAP může například obsahovat autentizační přihlašovací doklady uživatele nebo ID relace. Tyto podrobné informace mohou být požadovány pro zpracování požadavku, nicméně nesouvisí přímo s volanou metodou. Pokud oddělíte tyto dvě části, dosáhnete těchto vylepšení:

- **Rozhraní metody je jednodušší.** Nemusíte například vytvářet verzi metody `GetEmployees()`, která jako parametry akceptuje jméno uživatele a heslo. Tyto informace jsou totiž předávány v záhlaví SOAP a metoda tudíž není tolik "přecpána".
- **Služba je flexibilnější.** Přidáte-li například službu autentizace pomocí záhlaví SOAP, máte možnost změnit, jak daná služba funguje, a které informace požaduje, aniž byste změnili rozhraní vašich webových metod. Podobně jako u všech typů programování jsou více flexibilní řešení vždy upřednostňována.

Element <Header> je nepovinný a povoluje, aby byl do záhlaví umístěn neomezený počet dceřiných elementů. Pokud chcete definovat nová záhlaví pro použití s webovou službou .NET, vytvořte třídy, které budou odvozeny ze `System.Web.Services.Protocols.SoapHeader`.

Představte si například, že chcete navrhnout lepší způsob pro podporu stavu ve webové službě. Místo pokusu použít cookie relace můžete rovnou předat ID relace v záhlaví každé SOAP zprávy. Následující sekce implementují tento návrh.

Vlastní záhlaví

Prvním krokem k implementaci tohoto návrhu je vytvoření vlastní třídy odvozené ze `SoapHeader`, která obsahuje informace, které chcete předat jako veřejné vlastnosti.

Zde uvádíme příklad:

```
public class SessionHeader : SoapHeader
```

```

{
    public string SessionID;
    public SessionHeader(string sessionID)
    {
        SessionID = sessionID;
    }
    // Pro automatickou deserializaci je požadován standardní konstruktor.
    public SessionHeader()
    {}
}

```

Třída `SoapHeader` je ve skutečnosti pouze kontejnerem dat, který může být serializován uvnitř nebo vně elementu `<Header>` SOAP zprávy. Vlastní `SessionHeader` přidává proměnnou řetězce `SessionID` s klíčem `relace`.

Provázání záhlaví s webovou službou

Chcete-li použít `SessionHeader` ve webové službě, potřebujete pro záhlaví vytvořit veřejnou členskou proměnnou ve webové službě, jak to vidíte zde:

```

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

```

Když budete pro tuto webovou službu budovat třídu proxy, bude automaticky obsahovat vlastnost `CurrentSessionHeader`. Pomocí této vlastnosti můžete přečíst nebo nastavit záhlaví relace. Definice vlastní třídy `SessionHeader` je rovněž přidána do souboru třídy proxy.

Záhlaví jsou provázány s jednotlivými metodami prostřednictvím atributu `SoapHeader`. Chcete-li například použít službu `SessionHeader` ve webové metodě s názvem `DoSomething()`, použili byste atributy `WebMethod` a `SoapHeader` následujícím způsobem:

```

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

```

Všimněte si, že atribut `SoapHeader` je pojmenován podle veřejné členské proměnné, do které má .NET uložit SOAP záhlaví. V metodě `DoSomething()` atribut `SoapHeader` nařizuje ASP.NET vytvořit nový objekt `SessionHeader` použitím informací záhlaví, které byly získány od klienta a uložit jej do veřejné členské proměnné `CurrentSessionHeader` webové služby. ASP.NET používá reflexi, aby v běhovém režimu našel tuto členskou proměnnou. Pokud není přítomna, vznikne chyba. Atribut `SoapHeader` může také akceptovat jmenovitě uvedenou vlastnost `Direction`. Tato vlastnost specifikuje, zdali bude SOAP záhlaví zasláno z webového klienta webové služby, z webové služby webovému klientovi nebo oběma směry.

Následující příklad ilustruje, jak lze použít záhlaví relace pro vytvoření jednoduchého systému pro ukládání stavu. Nejprve webová metoda `CreateSession()` umožní klientovi inicializovat novou relaci. Poté je vygenerováno nové ID relace pro nový objekt `SessionHeader` a je vytvořena nová kolekce `Hashtable` v kolekci `Appli-`

cation, která je registrována pod tímto ID relace. Protože ID relace používá GUID, je statisticky zaručeno, že u všech uživatelů bude unikátní.

```
[WebMethod()]
[SoapHeader("CurrentSessionHeader", Direction=SoapHeaderDirection.Out)]
public void CreateSession()
{
    CurrentSessionHeader = new SessionHeader(Guid.NewGuid().ToString());
    // Od teď budou veškerá data session indexována pod tímto klíčem.
    Application[CurrentSessionHeader.SessionID] = new Hashtable();
}
```

Tato kolekce Hashtable bude použita pro uschovávání doplňkových informací relace. Není to právě nejlepší přístup – například kolekce Application není sdílena mezi počítači webové farmy, není perzistentní v případě restartu webové aplikace, a není rozšiřitelná pro velký počet uživatelů. Tento přístup však můžete snadno rozšířit tak, aby používal kombinaci databáze a cachování. Toto řešení by bylo mnohem více rozšiřitelnější, nehledě na to, že by používalo stejný systém záhlaví relace, jako je systém uvedený v našem příkladě.

Určitě jste si všimli, že metoda CreateSession() používá příkaz SoapHeaderDirection.Out, protože vytváří záhlaví a odesílá jej zpět klientovi. Nyní následuje velmi zajímavá část – když klient obdrží vlastní záhlaví, je toto uloženo do vlastnosti CurrentSessionHeader třídy proxy. Nejlepší na tom je, že počínaje tímto okamžikem, kdykoliv klientská aplikace zavolá metodu webové služby, která vyžaduje záhlaví, je toto záhlaví posláno společně s požadavkem. V podstatě – když klient používá stejnou třídu proxy, jsou záhlaví přenesena automaticky a systém řízení relace je tak zcela transparentní. Chcete-li si to otestovat, musíte do webové služby přidat dvě další metody. První metoda, SetSessionData(), akceptuje sadu dat a ukládá je do slotu Application aktuální uživatelské relace.

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

POZNÁMKA V tomto příkladě nemusíte kolekci Application uzamykat. Je to proto, že neexistují dva klienti, kteří by používali stejné ID relace, takže neexistuje možnost, že by se dva uživatelé pokusili současně změnit tento slot kolekce Application.

Dále můžete použít metodu GetSessionData() pro získání sady dat aktuální relace uživatele a jeho vrácení:

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


Samozřejmě – kdybyste vytvářeli skutečnou implementaci tohoto modelu, informace relace byste neukládali do stavu aplikace, protože stav aplikace není dostatečně robustní a rozšiřitelný. (Problémy se stavem aplikace jsou probrány v kapitole 6.) Místo toho byste pravděpodobně uložili informace do databáze a cachovali je v datové cache (viz kapitolu 11) pro jejich rychlé opětovné získání.

Použití webové služby, která používá vlastní záhlaví

Pokud webová metoda vyžaduje SOAP záhlaví, nemůžete ji otestovat prostřednictvím jednodušších protokolů HTTP GET nebo HTTP POST. To znamená, že kód nemůžete otestovat na testovací stránce prohlížeče. (Na této stránce se dokonce ani neobjeví tlačítko Invoke.) Místo toho si musíte vytvořit jednoduchého klienta.

Následující kód demonstruje testovací ukázkou. Kód vytvoří relace (a současně obdrží SOAP záhlaví), na serveru uloží novou prázdnou sadu dat a poté jej získá.

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

SOAP zpráva, pomocí které byla vyvolána CreateSession(), je podobná zprávě z předchozích příkladů:

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

Zpráva o odpovědi obsahuje SOAP záhlaví:

```
<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>
```

Následující volání metody již automaticky obsahují SOAP záhlaví, jak vidíte zde:

```
<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>
```

Výsledkem je webová služba, která poskytuje alternativní mechanismus stavu relace, který místo méně spolehlivých HTTP cookies používá SOAP záhlaví. SOAP záhlaví můžete ovšem použít i pro vytvoření mnoha rozšíření webové služby. V další kapitole si ukážeme, jak vám tato záhlaví umožní používat nově se objevující standardy webových služeb s komponentami Web Services Enhancements od Microsoftu.

WSDL

WSDL (Web Service Description Language) je jazyk založený na XML, který se používá k popisu veřejného rozhraní webové služby a komunikačních protokolů, které podporuje. WSDL dokument je v podstatě kontrakt, který sděluje klientovi vše, co potřebuje vědět, aby mohl spolupracovat s webovou službou. WSDL dokument má v podstatě stejnou úlohu jako typová knihovna u COM komponenty. (Ve světě .NET neexistuje přímá analogie s typovou knihovnou, protože všechny informace popisného typu, které potřebujete, jsou vloženy jako metadata do zkompilované assembly.)

SOAP poskytuje možnost komunikovat s webovou službou. Nicméně vám neříká, jak máte naformátovat své zprávy. Bez WSDL by bylo vaším úkolem zdokumentovat a vysvětlit XML formát, který vaše webové služby očekávají v obálce SOAP. Po lokalizaci webové služby by vývojáři klienta museli těmito informacím porozumět a adekvátně zpracovat SOAP zprávy požadavku a odpovědi. Pokud by přístup ke každé nové webové službě vyžadoval takovouto intervenci vývojáře, byl by posun směrem k webovým službám rozhodně zpomalen.

WSDL vyplňuje mezery tím, že popisuje podporované protokoly a očekávané formáty zpráv, které používá webová služba. Síla WSDL spočívá v tom, že není svázán se žádnou specifickou platformou nebo objektovým modelem. Je to jazyk XML, který poskytuje rozhraní pro webové služby v rámci všech platform.

Kompletní WSDL standard najdete na <http://www.w3.org/TR/wsdl>. Tento standard je dost komplexní, nicméně jeho podkladová logika je před vývojáři schována v programování ASP.NET, stejně jako jsou ovládací prvky ASP.NET odděleny od komplikovaných detailů HTML značek a atributů.

TIP V závislosti na typu webových služeb, které vytváříte, možná nebudete ani potřebovat zobrazit WSDL informace, protože se spokojíte s tím, že tuto úlohu přenecháte .NET, které pro vás automaticky vygeneruje tyto WSDL informace. Pokud však potřebujete podporovat klienty nějaké třetí strany, nebo pokud máte v plánu použít vývojářské techniky typu contract-first (jsou popsány v sekci "Implementace existujícího kontraktu"), budete potřebovat hlubší znalosti.

Zobrazování WSDL pro webovou službu

Jakmile jste vytvořili webovou službu, můžete snadno přimět ASP.NET k tomu, aby vám vygenerovalo odpovídající WSDL dokument. Potřebujete pouze požádat o .asmx soubor webové služby a na konec URL adresy přidat ?WSDL. (Další možností je kliknout v testovací stránce prohlížeče na odkaz Service Description, který směřuje na tuto URL adresu.) Obrázek 33-7 zobrazuje část WSDL dokumentu, který je dostupný pro webovou službu EmployeesService, vytvořenou v předchozí kapitole.



Obrázek 33-7. WSDL dokument pro EmployeesService.

WSDL dokument je velmi důležitý, protože umožňuje návrhářům programovacích pracovních rámců (jako je třeba .NET) vytvářet nástroje, které mohou programově generovat třídy proxy. Pokud přidáte webovou referenci ve Visual Studiu (nebo pokud použijete wsdl.exe), odkážete se tím na WSDL dokument webové služby. (Pokud se jedná o webovou službu .NET, můžete ušetřit jeden krok, jestliže se přímo odkážete na .asmx soubor webové služby. Oba nástroje jsou totiž dostatečně inteligentní na to, aby přidaly ?WSDL na konec dotazovacího řetězce pro získání WSDL dokumentu pro webovou službu .NET.) Nástroj si poté WSDL dokument pečlivě prohlédne a vytvoří třídu proxy, která bude používat stejné metody, parametry a datové typy. Jiné jazyky a programovací platformy rovněž poskytují nástroje, které pracují obdobným způsobem.

POZNÁMKA WSDL dokument obsahuje informace pro komunikaci mezi webovou službou a klientem. Neobsahuje žádné informace, které by se jakkoliv týkaly kódu nebo implementace metod vaší webové služby – tyto informace totiž nejsou nezbytně nutné a mohly by ohrozit bezpečnost. Zapamatujte si, že když přidáváte webovou referenci, vše co potřebujete, je pouze WSDL dokument. Ani Visual Studio, ani nástroj wsdl.exe nemají schopnost přímo prozkoumat kód webové služby.

WSDL dokumenty často bývají extrémně dlouhé, a proto je s nimi více práce než v případě jednoduché SOAP zprávy. V následujících několika sekcích si podrobněji ukážeme vzorový WSDL dokument pro EmployeesService.

Základní struktura

WSDL dokumenty sestávají z pěti hlavních elementů, které se vzájemně kombinují a popisují webovou službu. První tři jsou abstraktní a definují výměnu zpráv. Poslední dva jsou konkrétní, definují protokol a týkají se informací.

Tři abstraktní elementy, `<types>`, `<message>` a `<portType>`, se vzájemně kombinují a definují rozhraní webové služby. To znamená, že definují metody, parametry a vlastnosti webové služby. Dva konkrétní elementy, `<binding>` a `<port>`, se kombinují a poskytují nejenom protokol (SOAP přes HTTP), ale také adresní informace (URI) webové služby. Oddělení definičních elementů zpráv od informací o umístění a protokolu poskytuje flexibilitu, která umožňuje opakované využití společné sady zpráv a datových typů u různých protokolů. To sice činí WSDL dokumenty poněkud komplikovanějšími, nicméně jim poskytuje do budoucna neomezenou flexibilitu. V budoucnu by například mohly být WSDL dokumenty používány pro popis vzájemné interakce webových služeb SMTP, FTP nebo jiných, zcela odlišných, síťově založených protokolů.

Specifikace WSDL 1.1 přichází se SOAP přes HTTP, HTTP GET, HTTP POST a rozšíření MIME. ASP.NET podporuje všechno kromě rozšíření MIME. Protože standard WSDL je obvykle implementován pomocí SOAP přes HTTP, a protože je to výchozí nastavení ASP.NET, tato kapitola bude zaměřena na WSDL ve vztahu k SOAP a HTTP. SOAP je jednoznačně dominantní a zdá se, že v této chvíli nemá žádného skutečného soupeře. V současnosti to vypadá, že Microsoft před podporou MIME upřednostňuje podporu nezpracovaného SOAP přes TCP, dále podporu přenosového protokolu známého jako DIME (Direct Internet Message Encapsulation) a také projevuje větší zájem ohledně vytváření HTTP modifikací pro strukturu požadavku a odpovědi.

Element `<definitions>` je kořenovým elementem WSDL dokumentu. Je to místo, kde se definuje většina jmenových prostorů. Uvnitř elementu `<definitions>` se nachází pět hlavních elementů, z nichž jeden, `<message>`, se obvykle vyskytuje vícekrát.

Zde je základní struktura WSDL dokumentu:

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

Následuje přehled hlavních elementů WSDL dokumentu:

- **Types.** Tato sekce je místem, kde jsou definovány všechny datové typy webové služby. Patří sem rovněž vaše vlastní datové typy a formáty zpráv.
- **Messages.** Tato sekce poskytuje podrobné informace o zprávách o požadavku a odpovědi, které jsou používány při komunikaci s webovou službou.
- **PortType.** Tato sekce seskupuje zprávy do dvojice vstupních a výstupních zpráv. Každá dvojice reprezentuje metodu.
- **Binding.** Tato sekce poskytuje informace o transportních protokolech podporovaných webovou službou.

- **Service.** Tato sekce poskytuje koncové body (adresy URI) webové služby.

V následujících sekcích se budeme těmto elementům věnovat podrobněji.

Sekce types

Sekce <types> je místem, kde jsou definovány datové typy webové služby. Element <types> je v zásadě vložené schéma XML, přičemž všechny datové typy, které jsou definovány v rámci standardu Schéma XML, jsou validní. V případě potřeby můžete prostřednictvím rozšíření přidat další typové systémy.

Ve webové službě .NET je každá zpráva definována jako komplexní typ. Definice komplexního typu popisuje název metody, její parametry, nejnižší a nejvyšší počet možných výskytů elementu a datové typy. Vezměte si například metodu GetEmployeesCount(). Zpráva o požadavku nepotřebuje žádná data a je definována pomocí této syntaxe schématu XML:

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

Zpráva o odpovědi vrací celé číslo (typ int ze standardu Schéma XML) a je definována takto:

```
<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>
```

Sekce <types> bude v obvyklém případě velmi dlouhá, protože definuje dva komplexní typy pro každou webovou metodu.

Mnohem zajímavějším příkladem je webová služba, která vrací vlastní objekt. Vezměte si například tuto verzi metody GetEmployees(), která vrací pole objektů EmployeeDetails:

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

Podíváte-li se na sekci <types> této webové služby, zjistíte, že zpráva požadavku je nezměněna:

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

Odpověď se však odkazuje na další komplexní typ s názvemArrayOfEmployeeDetails:

```
<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 je komplexní typ, který je generován automaticky. Představuje seznam žádného nebo více objektů EmployeeDetails, jak vidíte zde:

```

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

```

Třída dat EmployeeDetails je rovněž definována jako komplexní typ v sekci <types>. Sestává z EmployeeID, FirstName, LastName a TitleOfCourtesy, jak vidíte zde:

```

<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>

```

Více o tom, jak komplexní typy fungují ve webových službách, se dozvíte později v této kapitole (v sekci "Přípůsobení zpráv SOAP").

Další položka, která se objevuje v sekci <types>, je definice jakéhokoliv vámi použitého záhlaví SOAP. Například test stavu webové služby, který byl v této kapitole již vytvořen, definuje následující typ, který reprezentuje data záhlaví relace:

```

<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>

```

Sekce messages

Zprávy reprezentují informace, vyměňované mezi metodou webové služby a klientem. Pokud požádáte jednoduchou webovou službu o ceny akcií, ASP.NET odešle zprávu a webová služba vrátí odlišnou zprávu. Definiční pro tyto zprávy najdete v sekci <message> WSDL dokumentu. Zde uvádíme příklad:

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

V tomto příkladě si povšimněte, že ASP.NET vytvoří zprávu GetEmployeesCountSoapIn a také GetEmployeesCountSoapOut. Pojmenování zpráv sice podléhá konvenci, nicméně je zdůrazněno, že pro vstup (zaslání parametrů a zavolání metody webové služby) a výstup (získání a vrácení hodnoty z metody webové služby) je požadována samostatná zpráva (resp. samostatné elementy).

Data použitá v těchto zprávách, jsou definována na základě informací v sekci <types>. Například zpráva požadavku GetEmployeesCountSoapIn používá zprávu GetEmployeesCount, která je definována jako prázdný komplexní typ v sekci <types>.

Sekce portType

Informace v sekci <portType> WSDL dokumentu poskytují katalog funkcionalit, které jsou dostupné ve webové službě. Na rozdíl od elementu <message>, o němž jsme právě hovořili, a který obsahuje nezávislé elementy jak pro vstupní, tak i výstupní zprávy, tyto operace jsou vzájemně svázány do seskupení požadavku a odpovědi. Název operace je názvem metody. <portType> je kolekcí operací, jak vidíte zde:

```
<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>
```

V kódu je vidět element <documentation> s informacemi, které jsou přidávány prostřednictvím vlastnosti Description atributu WebMethod.

POZNÁMKA

Existují čtyři typy operací – tzv. one-way (jednosměrná), request-response (požadavek-odpověď), solicit-response (vyžadovaná odpověď) a notification (oznamování). Současná WSDL specifikace definuje vázání pouze u operací typu one-way a request-response. Další dva typy mohou mít vázání definované prostřednictvím rozšíření vázání (binding extensions). Třetí a čtvrtý typ operace představují opak prvního a druhého typu operace – liší se pouze v tom, zdali koncový bod v dotazu leží na přijímacím, nebo na odesílacím konci výchozí zprávy. HTTP je obousměrným protokolem, takže jednosměrné operace budou fungovat pouze s MIME (který ovšem není podporován v ASP.NET) nebo s nějakým jiným, vlastním rozšířením.

Sekce binding

Elementy <binding> spojují abstraktní formát dat s konkrétním protokolem používaným pro přenos v rámci internetového spojení. WSDL dokument doposud specifikoval datové typy používané u různých informací, požadované zprávy používané pro operace, a strukturu každé zprávy. Prostřednictvím elementu <binding> specifikuje WSDL dokument komunikační protokol nižší úrovně, který můžete použít pro komunikaci s webovou službou. Současně jej spojuje s elementem <operation> ze sekce <portType>.

Ačkoliv zde nebudeme probírat všechny podobnosti kódování SOAP, zde je příklad, který definuje, jak by měla komunikace SOAP fungovat v případě metody GetEmployeesCount() z 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>
```

Pokud vaše metoda používá SOAP záhlaví, tyto informace budou přidány do elementu <header>. Zde uvádíme příklad metody CreateSession() z vlastní webové služby stavu, která byla v této kapitole vytvořena. Metoda CreateSession() nevyžaduje, aby klient odeslal záhlaví, nicméně metoda záhlaví vrací. Výsledkem je, že pouze výstupní zpráva obsahuje referenci na záhlaví:

```
<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>

```

Pokud vaše webová služba podporuje SOAP 1.2, naleznete zduplikovanou sekci <binding> s podobnými informacemi:

```

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

```

Zapamatujte si, že webové služby .NET standardně podporují jak SOAP 1.1, tak i SOAP 1.2. Toto chování ovšem můžete změnit pomocí konfiguračních souborů, jak už bylo uvedeno v této kapitole.

Sekce service

Sekce <service> definuje pro vaši webovou službu vstupní body ve formě jednoho nebo více elementů <port>. Každý <port> poskytuje informaci o adrese nebo URI. Zde je příklad z WSDL pro webovou službu 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>

```

Sekce <service> obsahuje také element <documentation> s vlastností Description atributu WebService (pokud je nastaven).

Implementace existujícího kontraktu

Od chvíle, kdy se poprvé objevily webové služby, se poněkud polemizovalo o správném způsobu, jak je vytvářet. Někteří vývojáři zastávají stanovisko, že nejlepším přístupem je používat platformy jako je .NET, které vytváří abstrakce pro podkladové detaily. Tito vývojáři chtějí pracovat s pracovním rámcem vyšší úrovně se vzdáleným voláním procedur. Vývojáři XML naopak argumentují, že na celý systém by mělo být pohlíženo z pohledu předávání zpráv XML. Jsou přesvědčeni o tom, že prvním krokem v jakékoliv aplikaci webové služby by mělo být ruční vytvoření WSDL kontraktu.

Podobně jako u jiných kontroverzních polemik leží řešení pravděpodobně někde uprostřed. Vývojáři aplikací pravděpodobně nebudou nikdy vytvářet WSDL kontrakty ručně – tento postup je příliš zdoluhavý a náchylný k chybám. Na druhé straně – vývojáři, kteří potřebují používat webové služby v širších meziplatformových scénářích, se budou muset zaměřit na výchozí XML reprezentaci svých zpráv a budou muset používat techniky jako je serializace atributů XML (tyto techniky budou popsány v další sekci), aby se drželi správného schématu.

.NET 1.x bylo až nestydatě orientované na vzdálené volání procedur. To značně omezovalo možnosti vývojářů se dostat pod fasádu webové služby a upravovat detaily nižší úrovně. Nový .NET 2.0 tato omezení řeší větší podporou postupů zaměřených na XML. Příkladem může být tzv. vývoj contract-first.

Ve scénářích webových služeb, které jste doposud viděli, je nejprve vytvořen kód webové služby. ASP.NET na požádání vygeneruje odpovídající WSDL dokument. Prostřednictvím .NET 2.0 však můžete tento problém uchopit z opačného konce. To znamená, že můžete vzít existující WSDL dokument a vložit jej do nástroje wsdl.exe příkazového řádku, čímž vytvoříte základní kostru webové služby. Všechno, co potřebujete, je pouze přepínač /serverInterface.

Například k vytvoření definice třídy pro webovou službu EmployeesService můžete použít následující příkaz (tento příkaz by měl být na jednom řádku, zde v knize je rozdělen na dva řádky):

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

Tím získáte následující rozhraní:

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

Toto rozhraní pak můžete implementovat v další třídě, čímž přidáte kód webové služby k metodě GetEmployees(). Pokud začínáte pracovat s ovládacím prvkem WSDL poprvé, ujistěte se, že implementace vaší webové služby mu přesně odpovídá.

POZNÁMKA Toto rozhraní není až tak jednoduché. Abyste se ujistili, že vaše rozhraní přesně odpovídá WSDL, přidává .NET mnoho atributů, které specificky nastaví detaily jako jmenné prostory, kódování SOAP či návyky elementů XML. Díky tomu je rozhraní "přecpané", nicméně základní struktura vypadá tak, jak je uvedeno zde.

Tento trik můžete rovněž použít pro webovou službu třetí strany. Je možné, že budete chtít vytvořit svoji vlastní verzi webové služby, která bude sledovat akcie na serveru XMethods.net. V takovém případě asi budete chtít zajistit, aby klienti mohli volat vaši webovou metodu bez toho, aby potřebovali získat nový WSDL dokument nebo bez toho, aby museli provést opětovnou kompilaci.

To můžete zajistit tak, že vygenerujete a implementujete rozhraní, které bude přesně odpovídat původnímu rozhraní:

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

Samozřejmě – k tomu, aby vaše webová služba byla skutečně kompatibilní, váš kód bude muset splňovat několik předpokladů, které nejsou specifikovány ve WSDL dokumentu. Tyto detaily mohou obsahovat informace o tom, jak parsujete řetězce, jak nakládáte s neplatnými daty, jak zpracováváte výjimky atd.

Není pravděpodobné, že vývoj typu contract-first (prvně vyvinout kontrakt) nahradí jednodušší model typu class-first (prvně vyvinout třídu). Je to však užitečná funkce pro vývojáře, kteří se potřebují držet již existujících WSDL kontraktů, zejména v meziplatformovém scénáři.

Přizpůsobování zpráv SOAP

V mnoha případech si nemusíte dělat starosti s detaily serializace SOAP. Ke štěstí vám bude stačit vytváření a používání webových služeb s použitím infrastruktury, kterou poskytuje .NET. V některých případech však možná budete potřebovat rozšířit své webové služby tak, aby používaly vlastní typy, nebo aby serializovaly vaše typy na specifický formát XML (pro dosažení meziplatformové kompatibility). V následujících sekcích si ukážeme, jak ovládat tyto detaily.

Serializace komplexních datových typů

Jak jste se již dozvěděli v předchozí kapitole, specifikace SOAP podporuje všechny datové typy, které jsou definovány standardem schéma XML. Tyto datové typy jsou považovány za jednoduché typy. SOAP však podporuje i komplexní typy, které odpovídají strukturám vybudovaným z uspořádání jednoduchých typů. Komplexní typy můžete použít jako návratovou hodnotu webové metody nebo jako parametr. Pokud však webová metoda vyžaduje parametry komplexního typu, můžete s nimi pracovat pouze prostřednictvím SOAP. Jednodušší mechanismy jako HTTP GET a HTTP POST nebudou funkční a testovací stránka prohlížeče vám neumožní zavolat příslušnou webovou metodu.

V knize jste již použili jednoho zástupce komplexních typů – sadu dat (DataSet). Když zavoláte metodu GetEmployees() ve webové službě EmployeesService, .NET vrátí XML dokument, který popisuje schéma sady dat a jeho obsah. Zde je uvedena část SOAP zprávy odpovědi:

```
<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">
          <!-- Schema omitted. -->
        </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>

```

U webových služeb .NET můžete použít své vlastní třídy. V takovém případě, když vybudujete proxy, kopie vlastní třídy bude automaticky přidána ke klientovi (v jazyku, který odpovídá klientovi).

Proces konverze objektů na XML je znám jako serializace; proces rekonstrukce objektů z XML je znám jako deserializace. Komponenta, která provádí serializaci, je třída `System.Xml.Serialization.XmlSerializer`. Tuto třídu byste ovšem neměli zaměňovat se třídami serializace, jako například `BinaryFormatter` a `SoapFormatter`, o nichž jsme hovořili v kapitole 13. Tyto třídy provádějí serializaci specifickou pro .NET, která pracuje s proprietárními objekty .NET (pokud jsou označeny atributem `Serializable`). Na rozdíl od `BinaryFormatter` a `SoapFormatter` pracuje `XmlSerializer` s jakoukoliv třídou, nicméně je mnohem více limitován než `BinaryFormatter` a `SoapFormatter` a může extrahovat pouze veřejná data.

Chcete-li použít `XmlSerializer` a zasílat vaše vlastní objekty do webové služby a zpět, musíte mít na paměti několik následujících omezení:

- **Jakýkoliv kód vložený vámi bude v klientovi ignorován.** To znamená, že klientova kopie vlastní třídy nebude obsahovat metody, logiku konstruktoru, nebo logiku procedury vlastností. Tyto detaily budou automaticky vyjmuty.
- **Vaše třída musí mít výchozí nulový argument konstruktoru.** To umožní .NET vytvořit novou instanci tohoto objektu, když deserializuje SOAP zprávu, která obsahuje odpovídající data.
- **Vlastnosti pouze pro čtení nejsou serializovány.** Řečeno jinými slovy – jestliže má vlastnost pouze "accessor" `get` a nikoliv "accessor" `set`, nemůže být serializována. Podobně jsou ignorovány soukromé vlastnosti a soukromé členské proměnné.

Z tohoto je zřejmé, že serializace třídy na mezipatformový kód XML má přísná omezení. Pokud ve webové službě používáte vlastní třídy, je lepší je používat jako jednoduché datové kontejnery, než jako skutečné spoluúčastníky (participants) objektově orientovaného návrhu.

Tvorba vlastní třídy

Chcete-li vidět `XmlSerializer` v akci, musíte vytvořit vlastní třídu a webovou metodu, která ji používá. V následujícím příkladě použijeme databázovou komponentu, kterou jsme poprvé vytvořili v kapitole 8. Tato databázová komponenta nepoužívá odpojené objekty sady `dat`. Místo toho vrací výsledky dotazu prostřednictvím vlastní třídy `EmployeeDetails`.

Zde vidíte, jak vypadá třída `EmployeeDetails`, bez jakýchkoliv vylepšení týkajících se webové služby:

```
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;
    }
}
```

Třída `EmployeeDetails` používá procedury vlastností místo veřejných členských proměnných. I přesto ji můžete použít, protože konverzi automaticky provede `XmlSerializer`. Třída `EmployeeDetails` nemá standardní nulový parametr konstruktoru, a proto předtím, než ji budete moci použít ve webové metodě, musíte k ní přidat konstruktory, jak vidíte zde:

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

Nyní je třída `EmployeeDetails` připravena pro scénář webové služby. Pokud ji chcete vyzkoušet, můžete vytvořit webovou metodu, která vrátí pole objektů `EmployeeDetail`. Následující příklad prezentuje jednu takovou metodu – webovou metodu `GetEmployees()`, která volá metodu `EmployeeDB.GetEmployees()` v databázové komponentě. (Kompletní kód této metody najdete v kapitole 8.)

Zde je webová metoda, kterou potřebujete:

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

Generování proxy

Když generujete proxy (buď s použitím wsdl.exe, nebo přidáním webové reference), dostanete dvě třídy. První třída je třída proxy, která je používána pro komunikaci s webovou službou. Druhá třída je definice EmployeeDetails.

Je důležité pochopit, že klientská verze EmployeeDetails neodpovídá verzi na straně serveru. Klient v podstatě nemůže vidět úplný kód třídy EmployeeDetails na straně serveru. Místo toho přečte WSDL dokument, který obsahuje XML schéma pro třídu EmployeeDetails. Toto schéma jednoduše vyjmenovává všechny veřejné vlastnosti a pole (aniž by mezi nimi nějak rozlišovalo) a jejich datové typy.

Když klient vybuduje třídu proxy, .NET použije tyto WSDL informace pro vygenerování třídy EmployeeDetails na straně klienta. Pro každou veřejnou vlastnost nebo pole definice EmployeeDetails na straně serveru, .NET přidá odpovídající veřejnou vlastnost ke třídě EmployeeDetails na straně klienta.

Zde uvádíme kód, který je vygenerován pro třídu EmployeeDetails na straně klienta:

```
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; }
}

```

V tomto případě je verze na straně klienta dost podobná verzi na straně serveru, protože verze na straně serveru neobsahuje příliš velké množství kódu. Jediným skutečným rozdílem (kromě přejmenování soukromých polí) je chybějící nestandardní konstruktor. Obecným pravidlem je, že verze na straně klienta neuchovává žádné nestandardní konstruktory, ani libovolný kód v proceduře vlastností nebo konstruktorů, žádné metody, a ani soukromé členy.

POZNÁMKA Verze datové třídy na straně klienta vždy používá procedury vlastností, a to dokonce i tehdy, když původní verze na straně serveru používá veřejné členské proměnné. To vám dává schopnost svázat kolekce objektů `EmployeeDetails` na straně klienta na ovládací prvek mřížky.

Testování vlastní třídy webové služby

Další krok představuje napsání kódu, který zavolá metodu `GetEmployees()`. Protože klient má nyní definici třídy `EmployeeDetails`, je tento krok velmi snadný:

```

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

```

Zpráva o odpovědi obsahuje data zaměstnanců v elementu `<GetEmployeesResult>`. `XmlSerializer` standardně vytváří strukturu dceřiných elementů, která je založena na názvu třídy (`EmployeeDetails`) a veřejné vlastnosti nebo na názvech proměnných (`EmployeeID`, `FirstName`, `LastName`, `TitleOfCourtesy` atd.). Zajímavé je, že tato výchozí struktura se dost podobá XML kódu, který je používán při modelování sady dat, ale bez informací schématu.

Zde uvádíme poněkud zkrácený příklad zprávy odpovědi:

```

<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>

```

```
</EmployeeDetails>
</GetEmployeesResult>
...
</GetEmployeesResponse>
</soap:Body>
</soap:Envelope>
```

Když klient obdrží tuto zprávu, je odpověď XML je zkonvertována na pole objektů `EmployeeDetails` s použitím definice třídy `EmployeeDetails` na straně klienta.

Přizpůsobení XML serializace s atributy

Občas můžete chtít přizpůsobit XML reprezentaci vlastní třídy. Tento přístup je nejužitečnější ve scénářích meziplatformového programování, kdy klient očekává XML v určité podobě. Můžete mít například již existující schéma, které očekává `EmployeeDetails` pro použití atributu `EmployeeID` místo vloženého elementu `<EmployeeID>`. .NET má k dispozici jednoduchý způsob, jak aplikovat tato pravidla, a sice s použitím atributů. Základní myšlenkou je, že na vaše datové třídy (jako je například `EmployeeDetails`) aplikujete atributy. Když `XmlSerializer` vytvoří SOAP zprávu, přečte tyto atributy a použije je pro úpravu XML, který vygeneruje.

Jmenný prostor `System.Xml.Serialization` obsahuje množství atributů, které mohou být použity k řízení podoby XML. Existují dvě sady atributů – jedna, v níž jsou atributy pojmenovány stylem `XmlXxx` a druhá, v níž jsou atributy pojmenovány stylem `SoapXxx`. To, které atributy použijete, záleží na tom, jakým způsobem jsou parametry kódovány.

Jak již bylo v této kapitole uvedeno, existují dva typy serializace SOAP – kódování literal a kódování SOAP sekce 5. Atributy `XmlXxx` využijte tehdy, pokud používáte parametry ve stylu literal. Jedná se o tyto případy:

- Používáte-li webovou službu s výchozí kódováním. (Řečeno jinými slovy, pokud jste nezměnili kódování přidáním libovolných atributů.)
- Používáte-li pro komunikaci s webovou službou protokol HTTP GET nebo HTTP POST.
- Používáte-li atribut `SoapDocumentService` nebo `SoapDocumentMethod` s vlastností `Use`, která je nastavena na `SoapBindingUse.Literal`.
- Používáte-li samotný `XmlSerializer` (mimo webové služby).

Atributy `SoapXxx` využijete tehdy, když používáte kódovaný styl parametrů. Je to v následujících případech:

- Používáte-li atribut `SoapRpcService` nebo `SoapRpcMethod`.
- Používáte-li atribut `SoapDocumentService` nebo `SoapDocumentMethod` s vlastností `Use` nastavenou na `SoapBindingUse.Encoded`.

U člena třídy je možné současně použít oba atributy – jak `SoapXxx`, tak i `XmlXxx`. Který z nich bude nakonec použit, záleží na typu serializace, která je prováděna. Tabulka 33-1 uvádí většinu atributů, které jsou k dispozici. Většina atributů obsahuje velké množství vlastností. Některé vlastnosti jsou společné pro většinu atributů, jako například vlastnost `Namespace` (která je použita pro indikaci jmenného prostoru serializovaného XML) či vlastnost `DataType` (která je použita pro indikaci specifického datového typu schématu XML, který nemusí být typem, který by si `XmlSerializer` zvolil jako výchozí). Kompletní popis všech atributů a jejich vlastností naleznete v nápovědě MSDN.

Tabulka 33-1. Atributy, které řídí XML serializaci.

Xml atribut	SOAP atribut	Popis
XmlAttribute	SoapAttribute	Používán k vytvoření polí nebo vlastností do atributů XML namísto elementů.
XmlElement	SoapElement	Používán pro pojmenování XML elementů.
XmlArray		Používán pro pojmenování polí.
XmlIgnore	SoapIgnore	Brání tomu, aby byly pole či vlastnosti serializovány.
XmlInclude	SoapInclude	Používán ve scénářích dědičnosti. Můžete mít například pole nebo vlastnost, která jsou typována jako nějaká základní třída, která se ve skutečnosti odkazuje na některou odvozenou třídu. V tomto případě můžete použít XmlInclude, kterým specifikujete všechny typy odvozených tříd, které můžete použít.
XmlRoot		Používán pro pojmenování elementu nejvyšší úrovně.
XmlText		Používán pro serializaci polí přímo v textu XML bez elementů.
XmlEnum	SoapEnum	Používán pro pojmenování členů výčtu tak, aby se jejich název odlišoval od názvu výčtu.
XmlType	SoapType	Používán pro řízení názvů typů ve WSDL souboru.

Pokud chcete vidět, jak funguje SOAP serializace, můžete použít tyto atributy u EmployeeDetails. Zamyslete se například nad následující deklarací modifikované třídy, která používá několik atributů serializace:

```
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;}
}
}
```

A zde vidíte, jak bude vypadat serializovaná EmployeeDetails v SOAP zprávě:

```
<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>
```

TIP Chcete-li experimentovat s různými atributy serializace, můžete přímo použít třídu XmlSerializer. Pouze vytvořte instanci XmlSerializer a předejte typ objektu, který chcete serializovat, jako parametr konstruktoru. Poté můžete použít metodu Serialize() pro konverzi objektu na XML a zápis dat do datového proudu (nebo do objektu TextWriter). Pro přečtení dat XML z datového proudu můžete použít metodu Deserialize(), nebo můžete použít TextReader a opětovně vytvořit původní objekt. Můžete rovněž použít nástroj příkazového řádku s názvem xsd.exe, který je součástí .NET Frameworku a vygenerovat pomocí něj definici třídy C#, která bude založena na dokumentech schématu XML. Deklarace třídy bude automaticky obsahovat odpovídající atributy serializace.

Tento příklad má pouze jedno omezení. Ačkoliv můžete řídit způsob serializace objektu EmployeeDetails, nemůžete použít stejné atributy pro zformování elementů, které zaobalují seznam zaměstnanců. U tohoto kroku máte dvě možnosti. Můžete vytvořit vlastní třídu kolekce a použít na ni atributy serializace XML. Chcete-li však pokračovat s použitím obvyčejného pole, musíte přidat atribut XML, který je přímo aplikován na návratovou hodnotu webové metody, jak vidíte zde:

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

Když nyní zavoláte webovou metodu, získáte tento kód XML:

```

<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>

```

Pro konfiguraci detailů můžete udělat ještě mnohem více. Můžete například vložit atributy serializace XML těsně před parametry, čímž změníte požadované XML příchozích zpráv požadavku. Můžete rovněž použít atribut SoapDocument (o kterém jsme již mluvili), kterým změníte název a jmenný prostor XML elementu, který zaobaluje návratovou hodnotu vaší funkce (v našem případě je to funkce s názvem <GetEmployeesResponse>).

Sdílení typů

V .NET 1.x mohly nastat potíže, pokud více než jedna webová služba používala stejnou vlastní třídu. Mohli jste například zavolat metodu Store.GetOrder() z jedné webové služby, čímž jste získali objekt Order, který jste poté zaslali metodě Shipping.TrackOrder() z jiné webové služby. Problém je v tom, že když přidáte referenci na obě webové služby, získáte dvě kopie třídy dat objektu Order ve dvou odlišných jmenných prostorech. A přestože jsou tyto definice tříd ekvivalentní, nemůžete je navzájem zaměňovat. To znamená, že když získáte objekt z jedné webové služby, nemůžete jej předat druhé webové službě.

.NET 2.0 tento problém řeší prostřednictvím nové funkce sdílení typů. Pomocí této funkce vygenerujete jednu kopii třídy dat na straně klienta a použijete ji u všech kompatibilních webových služeb. Aby mohla být vaše třída dat považována za kompatibilní, musí splňovat tyto dva požadavky:

- Musí mít stejnou reprezentaci XML. Řečeno jinými slovy – XML schéma typu musí být identické.
- Musí mít stejný jmenný prostor XML. Odlišné jmenné prostory indikují odlišné dokumenty.

Další detaily už nejsou tak důležité. Tyto faktory například nemají vliv na vaši schopnost provádět sdílení typů:

- Umístění webové služby.
- Jazyk webové služby.
- Název třídy nebo vlastností (pokud použijete atributy XML pro serializaci, abyste se ujistili, že serializovaná forma je odpovídající).

Například – následující třída Employee na straně serveru se poněkud odlišuje od třídy EmployeeDetails, která byla uvedena v předchozí sekci, serializovaná forma je nicméně identická:

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

Tato třída sice vyhovuje prvnímu požadavku (identické schéma XML), nicméně nemusí vyhovovat požadavku druhému (stejný jmenný prostor). Máte dvě možnosti, jak zajistit, aby se třída nacházela ve stejném jmenném prostoru. První možností je použít v atributu WebService shodný jmenný prostor pro obě webové služby:

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

Obvykle to však není přístup, který chcete použít. Má za následek, že obě webové služby budou stejné. Lepší možností je použít unikátní jmenný prostor pro identifikaci sdílených datových struktur XML. Můžete to provést prostřednictvím atributu XmlRoot nebo XmlElement u tříd EmployeeDetails a Employee, jak vidíte zde:

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

Serializované XML nyní vypadá ve zprávě odpovědi následovně:

```
<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>
      </GetEmployeesResult>
    </GetEmployeesResponse>
  </soap:Body>
</soap:Envelope>
```

```
...
</GetEmployeesResponse>
</soap:Body>
</soap:Envelope>
```

Za předpokladu, že jste vyhověli těmto dvěma požadavkům, jste připraveni vygenerovat sdílený typ. V současnosti je to možné pouze s použitím nástroje `wsdl.exe` příkazového řádku s přepínačem `/sharetypes`. Musíte vložit adresy všech webových služeb, které používají stejný typ. Zde uvádíme příklad:

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

Pokud nejsou typy z nějakého důvodu identické, budete o problému informováni. V takovém případě bude třída proxy obsahovat více verzí třídy dat, například `EmployeeDetails`, `EmployeeDetails1` atd.

Přizpůsobení XML serializace pomocí `IXmlSerializable`

Atributy XML pro serializaci fungují dobře, pokud využijete výhody mapování "jeden k jednomu" mezi vlastnostmi a elementy XML nebo atributy. V některých situacích však vývojáři potřebují mnohem více flexibility pro vytvoření XML reprezentace typu, která by odpovídala specifickému schématu. Možná potřebujete změnit reprezentaci vašich datových typů, řídit pořadí elementů, nebo přidávat dodatečné informace (jako jsou například komentáře nebo datum, kdy byl dokument serializován). V některých případech je sice technicky možné použít atributy XML pro serializaci, nicméně to může znamenat vytvoření velmi nešikovného modelu třídy.

Naštěstí .NET 2.0 má k dispozici rozhraní `IXmlSerializable`, které můžete implementovat, abyste získali úplnou kontrolu nad vaším XML. Atribut `IXmlSerializable` existuje v .NET už od verze 1.0. Byl však používán jako proprietární způsob pro přizpůsobení serializace sady dat .NET a nebyl dostupný pro všeobecné použití. Nyní je plně podporován. `IXmlSerializable` definuje tři metody, které jsou uvedeny v tabulce 33-2.

Tabulka 33-2. Metody `IXmlSerializable`.

Metoda	Popis
<code>WriteXml()</code>	V této metodě zapisujete XML reprezentaci instance vašeho objektu pomocí <code>XmlWriter</code> . Tuto metodu potřebujete k tomu, aby vaše webová služba serializovala objekt a zaslala jej jako návratovou hodnotu.
<code>ReadXml()</code>	V této metodě čtete XML z <code>XmlReader</code> a generujete odpovídající objekt. Je možné, že tuto metodu nebudete vůbec potřebovat (v tom případě je bezpečné zavolat <code>NotImplementedException</code>). Budete ji však potřebovat, pokud budete muset deserializovat objekt, který vaše webová služba akceptuje jako vstupní parametr, nebo tehdy, pokud se rozhodnete rozmístit tuto vaši vlastní třídu u klienta.
<code>GetSchema()</code>	Tato metoda je zavržena a měli byste vrátit <code>null</code> . Pokud chcete docílit schopnosti vygenerovat schéma XML pro vaši třídu (které bude začleněno do WSDL dokumentu), musíte použít atribut <code>XmlSchemaProvider</code> . <code>XmlSchemaProvider</code> pojmenovává metodu vaší třídy, která vrací tzv. dokument schématu XML (XML schema document, XSD).

Třídy `XmlReader` a `XmlWriter` byly podrobně popsány v kapitole 12. Jejich použití je dost jednoduché. Zde uvádíme příklad vaší vlastní třídy, která zařizuje vygenerování vlastního XML:

```
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;
    }
    // (Konstruktory byly vynechány.)
}
```

TIP Ujistěte se, že jste v metodě `ReadXml()` kompletně přečetli celý dokument XML, včetně značek uzavíracího elementu. V opačném případě může .NET vyvolat výjimku při pokusu o deserializaci XML.

Pokud nyní vytvoříte následující webovou metodu:

```
[WebMethod()]
public EmployeeDetailsCustom GetCustomEmployee()
```

```
{
    return new EmployeeDetailsCustom(101, "Joe", "Dabiak");
}
```

Získáte toto XML:

```
<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>
```

POZNÁMKA Při použití `IXmlSerializable` jsou efektivní pouze ty atributy serializace, které použijete u metody a deklarace třídy. Atributy použité u jednotlivých vlastností a polí nemají žádný efekt. Můžete však použít reflexi .NET, zkontrolovat vaše vlastní atributy, a poté je použít pro úpravu značek XML, které vygenerujete.

Schémata pro vlastní datové typy

Jediným omezením v tomto příkladě je to, že klient nedokáže určit, jaké XML má vlastně očekávat. Podíváte-li se v tomto příkladě na sekci <types> ve WSDL dokumentu, uvidíte, že schéma je široce otevřené prostřednictvím elementu <any>. To umožňuje použití jakéhokoliv platného obsahu XML.

```
<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>
</s:sequence>
</s:complexType>
</s:element>
```

Na straně klienta můžete zacházet s daty jako s fragmentem XML – v tomto případě potřebujete napsat parsovací kód XML. Lepší způsob je však doplnit schéma XML pro vaši vlastní reprezentaci XML. K tomu potřebujete přidat do vaší třídy statickou metodu, která vrátí dokument schématu XML jako objekt `XmlQualifiedName`, jak vidíte zde:

```
public static XmlQualifiedName GetSchemaDocument(XmlSchemaSet xs)
{
    // Získej cestu do souboru schématu.
    string schemaPath = HttpContext.Current.Server.MapPath
        ("EmployeeDetails.xsd");
    // Získej schéma ze souboru.
    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);
}
```

TIP V tomto případě je dokument schématu získáván ze souboru. Lepšího výkonu však dosáhnete tehdy, pokud tento dokument uložíte do cache nebo jej vytvoříte programově.

Nyní potřebujete nasměrovat .NET ke správné metodě pomocí atributu `XmlSchemaProvider`:

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

ASP.NET po vygenerování WSDL dokumentu zavolá tuto statickou metodu. Poté do WSDL dokumentu přidá informace o schématu. Nezapomeňte však, že když vytváříte klienta, .NET vygeneruje datovou třídu, která odpovídá schématu, což znamená, že `EmployeeDetails` na straně klienta se bude poněkud odlišovat od verze na straně serveru. (V tomto případě bude mít třída `EmployeeDetails` na straně klienta kvůli uspořádání XML elementů vloženou třídu `Name`, což není to, co byste chtěli.)

Co tedy musíte udělat, pokud chcete mít stejnou verzi `EmployeeDetails` na straně klienta i serveru? Můžete například ručně změnit vygenerovaný kód třídy proxy, ačkoliv tato změna bude zrušena při každém opětovném vybudování proxy. Trvalejší možností je použití rozšíření importéru schématu, kterým se budeme zabývat v sekci "Rozšíření importéru schématu".

Vlastní serializace rozsáhlých datových typů

Jedním z důvodů, proč byste měli použít `IXmlSerializable`, je vytváření webových služeb, které zasílají velké množství dat. Představte si například, že chcete poslat velký blok binárních dat, který obsahuje obsah souboru. Můžete použít například tuto webovou službu:

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

Problémem je, že tento přístup předpokládá, že veškerá data ze souboru se najednou načtou do paměti ve formě bajtového pole. Ovšem pokud je soubor velký několik gigabajtů, spolehlivě to ochromí počítač. Lepším řešením je použít `IXmlSerializable` pro implementaci tzv. chunkingu. Tímto způsobem můžete poslat přes síť libovolně velké množství dat.

V následujících sekcích si ukážeme příklad, ve kterém se používá `IXmlSerializable` pro dramatické snížení zatížení, které je spojeno s posíláním velkého souboru.

Strana serveru

Prvním krokem je vytvoření signatury pro vaši webovou metodu. Aby tato strategie fungovala, musí webová metoda vracet třídu, která implementuje `IXmlSerializable`. V tomto příkladu je použita třída s názvem `FileData`. V ASP.NET musíte vypnout buffering, aby odpověď mohla být streamována po síti.

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

Nejnáročnější částí je implementace vlastní serializace ve třídě `FileData`. Základní myšlenkou je, že když na serveru vytvoříte objekt `FileData`, jednoduše specifikujete odpovídající název souboru. Když je objekt `FileData` serializován a je zavolána `IXmlSerializable.WriteXml()`, objekt `FileData` vytvoří `FileStream` a začne po jednotlivých blocích posílat binární data.

Zde uvádíme základní kostru třídy `FileData`:

```
[XmlRoot(Namespace="http://www.apress.com/ProASP.NET/FileData")]
[XmlSchemaProvider("GetSchemaDocument")]
public class FileData : IXmlSerializable
{
    // Jmenný prostor pro serializaci.
    const string ns = "http://www.apress.com/ProASP.NET/FileData";
    // Cesta na straně serveru.
    private string serverFilePath;
    // Po vytvoření FileData se ujisti, že soubor existuje.
    // Toto neposkytuje ochranu před dalšími problémy se čtením
    // souboru (jako jsou
    // nedostatečná práva, momentální uzamčení souboru jiným procesem,
    // a tak dále).
    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)
{
    // Získej cestu do souboru schématu.
    string schemaPath = HttpContext.Current.Server.MapPath("FileData.xsd");
    // Získej schéma ze souboru.
    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);
}
}

```

Všimněte si, že tato třída podporuje zápis souborových dat do XML, nikoliv však jejich čtení. Je to proto, že se díváte na serverovou verzi kódu. Ta odesílá objekty FileData, nicméně je nezískává.

V tomto příkladě chcete vytvořit XML reprezentaci, která rozdělí data do samostatných bloků (chunks), které budou zakódovány v Base64. Bude to vypadat následovně:

```

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

```

A zde je implementace WriteXml(), která vše provádí:

```

void IXmlSerializable.WriteXml(System.Xml.XmlWriter writer)
{
    // Otevři soubor (umožni, aby bylo možné jej otevřít
    //zároveň také jinými elementárními procesy).
    FileStream fs = new FileStream(serverFilePath, FileMode.Open,
    FileAccess.Read, FileShare.Read);
    // Zapiš název souboru.
    writer.WriteElementString("fileName", ns, Path.GetFileName(serverFilePath));
    // Zapiš velikost souboru (což je užitečné pro stanovení progresu.)
    long length = fs.Length;
    writer.WriteElementString("size", ns, length.ToString());
    // Spust' obsah souboru.
    writer.WriteStartElement("content", ns);
    // Přečti 4 KB vyrovnávací paměť a zapiš to (u trochu větších
    // bloků zakódovaných v 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);
        // Tato metoda explicitně kóduje data. Použijete-li jinou metodu,
        // je možné do XML datového proudu přidat neplatné znaky.
        writer.WriteBase64(fileBytes, 0, readBytes);
        writer.WriteEndElement();
        writer.Flush();
    }
    fs.Close();
    // Ukonči XML.
    writer.WriteEndElement();
}

```

Nyní můžete webovou službu dokončit. Zde uvedená metoda `DownloadFile()` vyhledává v natvrdo nadefinovaném adresáři soubor specifikovaný uživatelem. Metoda vytvoří nový objekt `FileData` s úplnou cestou a vrátí jej. V této chvíli se do procesu zapojí kód serializace `FileData`, který soubor přečte a začne jej zapisovat do proudu (stream) odpovědi.

```

public class FileService : System.Web.Services.WebService
{
    // Povol stahování pouze v tomto adresáři.
    string folder = @"c:\Downloads";
    [WebMethod(BufferResponse = false)]
    [SoapDocumentMethod(ParameterStyle = SoapParameterStyle.Bare)]
    public FileData DownloadFile(string serverFileName)
    {
        // Ujistí se, že uživatel specifikoval pouze název souboru

```

```
// (nikoli úplnou cestu).
serverFileName = Path.GetFileName(serverFileName);
// Získej úplnou cestu pomocí adresáře pro stahování souborů.
string serverFilePath = Path.Combine(folder, serverFileName);
// Vrat' data souboru.
return new FileData(serverFilePath);
}
}
```

Tuto metodu si můžete vyzkoušet pomocí testovací stránky prohlížeče, podívat se na kód XML a ověřit si, že data jsou rozdělena do bloků (chunks).

Strana klienta

Na straně klienta potřebujete způsob, kterým byste získávali data po jednotlivých blocích (chunk) a zapisovali je do souboru. Tuto schopnost získáte tak, že změníte třídu proxy, aby vracela vlastní typ `IXmlSerializable`. Do této třídy pak umístíte kód deserializace.

TIP Ve stejné třídě je možné implementovat jak kód serializace, tak i kód deserializace, přičemž tuto třídu je možné nabídnout klientovi a serveru ve formě komponenty. Obvykle je však lepší dodržovat striktní oddělení obou konců aplikace webové služby. Usnadní to aktualizaci klienta v případě nové verze.

Když vytvoříte třídu proxy, .NET se pokusí vytvořit vhodnou kopii třídy `FileData`. To se mu ovšem nepodaří. Bez informací o schématu se pokusí zkonvertovat vrácenou hodnotu na sadu dat. A dokonce i tehdy, když přidáte informace o schématu, .NET bude schopno vytvořit pouze reprezentaci třídy, která vystaví všechny detaily (název, velikost a obsah) prostřednictvím samostatných vlastností. Tato třída nebude mít chování typu "chunking", a pokusí se uložit do paměti všechno najednou.

Pokud chcete tento zdárný problém vyřešit, musíte třídu proxy ručně upravit. Pokud vytváříte webového klienta, nejprve potřebujete vygenerovat třídu proxy pomocí nástroje `wsdl.exe`, abyste měli k dispozici kód. Zde uvádíme změnu, kterou musíte provést:

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

Je jasné, že modifikace třídy proxy je křehkým řešením, protože při každém obnovení třídy proxy bude vaše změna odstraněna. Mnohem lepší volbou je implementovat rozšíření importéru schématu, což bude popsáno v následující sekci.

Zde je základní náčrt třídy `FileDataClient`:

```
[XmlRoot(Namespace="http://www.apress.com/ProASP.NET/FileData")]
public class FileDataClient : IXmlSerializable
{
```

```

private string ns = "http://www.apress.com/ProASP.NET/FileData";
// Umístění staženého souboru.
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();
}
}

```

Důležitým prvkem je statická vlastnost `ClientFolder`, která sleduje umístění, do něhož chcete uložit všechny stažené soubory. Tuto vlastnost musíte nastavit před spuštěním stahování, protože metoda `ReadXml()` používá uvedené informace při rozhodování, kde má vytvořit soubor. Vlastnost `ClientFolder` musí být statická, protože klient nemá možnost vytvořit a nakonfigurovat objekt `FileDataClient`, který chce použít. .NET místo toho automaticky vytvoří instanci `FileDataClient` a použije ji pro deserializaci dat. Použitím statické vlastnosti může klient nastavit tyto informace před spuštěním stahování, jak vidíte zde:

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

Kód deserializace má opačnou úlohu než kód serializace – prochází bloky (chunks) a zapisuje je do nového souboru. Zde uvádíme kompletní kód:

```

void IXmlSerializable.ReadXml(System.Xml.XmlReader reader)
{
    if (FileDataClient.ClientFolder == "")
    {
        throw new InvalidOperationException("No target folder specified.");
    }
    reader.ReadStartElement();
    // Získej původní název souboru.
    string fileName = reader.ReadElementString("fileName", ns);
    // Získej velikost (ne momentálně používanou).
    double size = Convert.ToDouble(reader.ReadElementString("size", ns));
    // Vytvoř soubor.
    FileStream fs = new FileStream(Path.Combine(ClientFolder, fileName),
        FileMode.Create, FileAccess.Write);
    // Přečti XML a zapiš soubor po blocích.
    byte[] fileBytes;

```

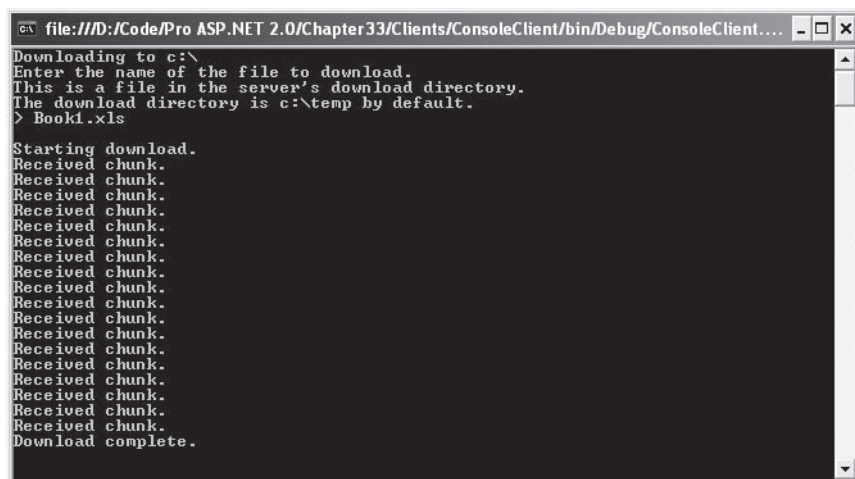


```
reader.ReadStartElement("content", ns);
double totalRead = 0;
while (true)
{
    if (reader.IsStartElement("chunk", ns))
    {
        string bytesBase64 = reader.ReadElementString();
        totalRead += bytesBase64.Length;
        fileBytes = Convert.FromBase64String(bytesBase64);
        fs.Write(fileBytes, 0, fileBytes.Length);
        fs.Flush();
        // Můžete hlásit progres, a sice tím, že zde zavoláte metodu.
        Console.WriteLine("Received chunk.");
    }
    else
    {
        break;
    }
}
fs.Close();
reader.ReadEndElement();
reader.ReadEndElement();
}
```

A zde uvádíme kompletní konzolovou aplikaci, která používá 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.");
}
```

Na obrázku 33-8 je zobrazen výsledek.



```
file:///D:/Code/Pro ASP.NET 2.0/Chapter33/Clients/ConsoleClient/bin/Debug/ConsoleClient...
Downloading to c:\
Enter the name of the file to download.
This is a file in the server's download directory.
The download directory is c:\temp by default.
> Book1.xls

Starting download.
Received chunk.
Received chunk.
Received chunk.
Received chunk.
Received chunk.
Received chunk.
Received chunk.
Received chunk.
Received chunk.
Received chunk.
Received chunk.
Received chunk.
Received chunk.
Received chunk.
Received chunk.
Received chunk.
Received chunk.
Received chunk.
Download complete.
```

Obrázek 33-8. Stahování velkého souboru pomocí tzv. "chunkingu".

Tento příklad je také možné najít na webu, pouze s několika menšími změnami (například klientské a serverové metody pro práci se souborem jsou zkombinovány do jediné třídy `FileData`).

Rozšíření importéru schématu

Jeden z klíčových principů návrhu orientovaného na službu (service-oriented design) je ten, že klient se serverem společně sdílí kontrakty, a nikoliv třídy. Tato úroveň abstrakce umožňuje klientům na zcela odlišných platformách pracovat se stejnou webovou službou. Webové služby sice zasílají shodný serializovaný kód XML, nicméně pro přípravu svých zpráv mohou volně používat různé programovací struktury (jako například třídy).

V některých případech možná budete chtít tato pravidla trochu pozměnit, abyste svým klientům umožnili pracovat s bohatými datovými typy. Například – možná budete chtít nabídnout klientovi a serveru svoji vlastní datovou třídu a umožnit jim zasílat nebo získávat instance této třídy prostřednictvím webové služby. .NET 2.0 vám toto umožňuje prostřednictvím nové schopnosti pojmenované jako rozšíření importéru schématu (schema importer extensions.).

TIP Použití vlastních datových typů si dvakrát rozmyslete. Nebezpečí tohoto přístupu tkví v tom, že vás může snadno navést k vytvoření proprietární webové služby. Ačkoliv bude vaše webová služba stále používat XML (které může být přečteno vždy a na libovolné platformě), v okamžiku, kdy začnete upravovat vaše XML, aby odpovídalo typům specifickým pro jednotlivé platformy, pro další klienty může být extrémně obtížné parsovat takové XML, nebo s ním provádět jakékoliv praktické úkony. Například většina klientů, kteří nepoužívají .NET, nemá k dispozici jednoduchý způsob, jak využít XML, které je vygenerováno pro sadu dat.

Než vytvoříte rozšíření importéru schématu, ujistěte se, že vaše služba používá atribut `XmlSchemaProvider` pro označení metody, která vrací informace schématu. Bez informací schématu nebude mít nástroj pro generování proxy potřebné informace pro identifikaci vašich vlastních datových typů, takže všechny vytvořené importéry schématu budou nepoužitelné.

U třídy `FileData` je schéma vytvořeno z následujícího souboru schématu:

```

<xs:schema id="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>

```

Nyní jste připraveni vytvořit importér schématu, který umožní klientovi rozpoznat tento datový typ.

Vytvoření importéru schématu se skládá z těchto kroků – je potřeba vytvořit rozšíření, které pak musíte zaregistrovat. Chcete-li vytvořit rozšíření, potřebujete vytvořit novou komponentu knihovny třídy (DLL assembly). V této assembly přidejte třídu odvozenou od `SchemaImporterExtension`.

Když generátor proxy narazí na komplexní typ (během generování třídy proxy), zavolá metodu `ImportSchemaType()` pro každé rozšíření importéru schématu, které je definováno v souboru `machine.config`. Každý importér schématu dokáže zkontrolovat jmenný prostor a schéma typu, a poté rozhodnout, zdali jej zpracuje mapováním typu XML na známý typ .NET.

Zde uvádíme příklad s `FileDataSchemaImporter`, který nakonfiguruje proxy tak, aby používala třídu `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
    {

```

```

        // Vyber možnost neobsloužit tento typ.
        return null;
    }
}
}

```

Toto je neobyčejně jednoduchý importér schématu, který provádí dvě věci:

- Nařizuje třídě proxy, aby pro tento typ používala třídu s názvem `FileDataClient`. To znamená, že třída proxy použije existující třídu a automaticky nevygeneruje třídu `FileData` na straně klienta (výchozí chování).
- Nařizuje generátoru třídy proxy, aby přidal ke jmennému prostoru `FileDataComponent` importovaný jmenný prostor. Je ovšem pouze na vás, abyste se ujistili, že assembly s třídou `FileDataComponent.FileData` je ve vašem projektu k dispozici.

Jakmile máte vytvořený importér schématu, musíte jej nainstalovat do globální cache pro assembly. Dejte mu silné jméno (použijte záložku `Signing` ve vlastnostech projektu) a poté jej kurzorem myši přetáhněte do adresáře `c:\[WinDir]\Assembly`, nebo použijte utilitu příkazového řádku `gacutil.exe`.

Jakmile je váš importér schématu bezpečně nainstalován do cache, pomocí Průzkumníka Windows najdete příznak veřejného klíče (public key token). Vyzbrojeni těmito informacemi můžete vašeho importéra schématu zaregistrovat v souboru `machine.config` pomocí těchto nastavení:

```

<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>

```

Atribut `type` je důležitou součástí nastavení. Ujistěte se, že následující kód jste použili na samostatném řádku:

```

<Namespace-qualified class name>, <Assembly name without the extension>,
<Version>, <Culture>, <Public key token>

```

Nyní jste připraveni použít váš importér schématu. Zkuste spustit `wsdl.exe` ve `FileService`:

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

Vygenerovaný kód proxy použije vámi specifikovaný název typu a bude obsahovat nový importovaný jmenný prostor. Nevytvoří však třídu `FileData`, ale bude používat vlastní verzi, kterou jste vytvořili v komponentě `FileData`.

A nakonec přidejte tuto vygenerovanou třídu proxy do vašeho klientského projektu. Nyní můžete stahovat soubory s podporou streamování typu `chunk-by-chunk`, kterou poskytuje třída `FileData`.

POZNÁMKA

V současné době používá importéry schématu pouze wsdl.exe. Importéry schématu nebudou používány, když vygenerujete webovou referenci pomocí Visual Studia. Očekává se, že v dalších verzích se toto změní.

Shrnutí

V této kapitole jste se blíže seznámili s nejdůležitějšími protokoly webových služeb – s protokolem SOAP a WSDL. SOAP je neobyčejně lehký protokol pro přenos zpráv. WSDL je flexibilní a rozšiřitelný protokol pro popis webových služeb. Společně zajišťují, že v následujících letech bude možné vytvářet a používat webové služby prakticky na jakékoliv programovací platformě. Tato kapitola se rovněž podrobně zabývala způsoby pro úpravu XML, který vrací vaše webová služba.

