

Kapitel 28

Webservices mit ASP.NET (ASMX)

In diesem Kapitel:

Webservices erstellen mit ASP.NET	582
Webservices in .NET-Anwendungen nutzen	590
Weitere Möglichkeiten	597

In das .NET Framework wurde die Unterstützung für die Nutzung und Bereitstellung von Webservices von Anfang an integriert. Die Bereitstellung von XML-basierten Webservices ist seit .NET 1.0 ein Teil von ASP.NET. Dabei werden XML-Webservices im Rahmen von ASP.NET in Form von *.asmx*-Dateien bereitgestellt. Daher sprechen Insider auch von ASMX synonym zu ASP.NET Webservices.

Die .NET Framework Class Library (FCL) enthält im Namensraum `System.Web.Services.Protocols` die Implementierung von SOAP. Das .NET Framework Version 1.0 weisen eine Implementierung für SOAP 1.1 auf; SOAP 1.2 gibt es zusätzlich ab dem .NET Framework 2.0. Außerdem werden WSDL 1.1 und als Transportprotokoll HTTP 1.1 unterstützt.

Das .NET Framework unterstützt inzwischen für ASP.NET-basierte Webservices alle vier SOAP-Arten (vgl. Abschnitte 5 und 7 in [W3C04]): *document/literal*, *document/encoded*, *rpc/literal* und *rpc/encoded*. Für die größtmögliche Interoperabilität sollten Sie nur *document/literal* verwenden, das der Standardeinstellung in .NET entspricht. Die SOAP-Art *document/literal* ist das einzige im Basic Profile (BP) 1.0 der Web Services Interoperability Organization (WS-I) [WSI01] vorgesehene Verfahren. Zur Nutzung anderer SOAP-Arten lesen Sie bitte [MSDN11].

Da die Beschreibung der Bereitstellung von Webservices alleine keinen Sinn machen würde, wird in diesem Kapitel ein Webserviceclient in Form einer Windows Forms-Anwendung gezeigt. Eine ASP.NET-Webanwendung kann aber auch Client eines Webservices sein. Dieses Szenario wird ebenfalls in diesem Kapitel behandelt.

HINWEIS

XML-Webservices nutzen als Transportformat zwar SOAP, das .NET Framework verwendet aber nicht die Klasse `System.Runtime.Serialization.Formatters.Soap.SoapFormatter`, weil diese nur die SOAP-Art *rpc/encoded* unterstützt. Das .NET Framework greift stattdessen für XML-Webservices auf den so genannten XML-Serialisierer (`System.Xml.Serialization.XmlSerializer`) zu.

Webservices erstellen mit ASP.NET

Webservices sind ein .NET-Anwendungstyp, der auf dem ASP.NET Runtime Host läuft, das heißt auf den IIS, die auf dem in Visual Studio integrierten ASP.NET Development Server oder einem anderen ASP.NET-fähigen Webserver betrieben werden müssen. Genau wie Web Forms-Anwendungen werden auch Webservices in Form von Dateien auf einem Webserver realisiert. Die Dateierweiterung eines Webservices lautet aber nicht *.aspx*, sondern *.asmx*. Die Abkürzung ASMX wird daher oft mit .NET-basierten Webservices gleichgesetzt. Unter dem Schlagwort ASMX 2.0 wird die Webservice-Server-Implementierung in .NET 2.0 zusammengefasst.

XML-Webservices sind grundsätzlich transportprotokollunabhängig. Die Nutzung von HTTP ist jedoch heutzutage die am weitesten verbreitete Transportform und auch die einzige, die im .NET Framework 1.x / 2.0 seitens ASMX unterstützt wird. Bei der Nutzung von HTTP liegt es nahe, ASP.NET als Runtime Host für Webservices einzusetzen. Grundsätzlich können alle ASP.NET-fähigen Webserver auch Webservices anbieten, also nicht nur die IIS, sondern auch Apache (mit Erweiterungen) und der mit Mono mitgelieferte Webserver XSP. Durch die in .NET 2.0 neu eingeführte Klasse `System.Net.HttpListener` ist es auf einfache Weise möglich, einen webservicefähigen HTTP-Server selbst zu implementieren.

Ein Webservice wird auf dem Webserver durch eine *.asmx*-Datei repräsentiert. Jede *.asmx*-Datei bietet genau eine Klasse nach außen an. (Sie kann weitere Klassen enthalten, die dann aber nicht per SOAP angesprochen werden können.) Jede Webserviceklasse kann beliebig viele, mit `<WebMethod()>` ausgezeichnete öffentliche Methoden enthalten, die als Webmethoden im Rahmen des Webservice nutzbar sind. Eine IIS-Webanwendung kann beliebig viele *.asmx*-Dateien enthalten. Im Client wird pro *.asmx*-Datei eine Proxyklasse erzeugt.

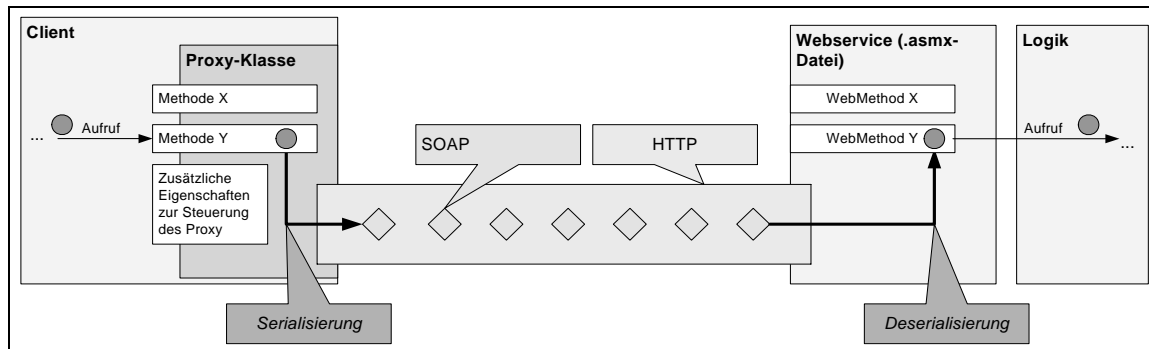


Abbildung 28.1 Architektur der ASP.NET-basierten Webservices

Modellalternativen

Genau wie bei Web Forms besteht die Wahl zwischen dem Ein-Datei- und dem Hintergrundcodemodell. Für Webservices ergibt das Hintergrundcodemodell jedoch keinen Sinn, denn eine abzutrennende Benutzerschnittstelle gibt es nicht. Vielmehr ist es sogar lästig, dass im Hintergrundcodemodell die *.asmx*-Datei nur aus einer Direktive (`@WebService`) mit Verweis auf die Hintergrundcodedatei besteht.

	Hintergrundcodemodell	Ein-Datei-Modell
Seitendirektive	<pre><%@ WebService Language="Sprache" CodeFile="~/App_Code/WWingsFlugplanService.sprache" Class="WWingsFlugplanService" %></pre>	<pre><%@ WebService Language="Sprache" Class=" WWingsFlugplanService" %></pre>
Implementierung	in separater Datei	direkt unter der Direktive

Vorgehensweise in Visual Studio 2008

Zum Anlegen eines Webservices wählen Sie unter *Datei/Neu/Website* den Eintrag *ASP.NET-Webdienst* (*ASP.NET Web Services*) oder Sie legen eine ASP.NET-Website an und fügen über das *Website*-Menü ein Element vom Typ *Webservice* hinzu. Sie können ein Webservice-Element auch zu einem bestehenden Web Forms-Projekt hinzufügen.

Als Programmiersprachen stehen wie bei Web Forms C#, Visual Basic und J# zur Verfügung. J# ist nicht im Visual Web Developer Express verfügbar.

Für den Code gilt:

- Die Webservice-Datei muss mindestens eine Klasse enthalten.
- Der Name dieser Klasse muss in der `@WebService`-Direktive hinter dem XML-Attribut `Class` genannt sein. Wichtig ist, dass – auch bei Visual Basic – die Groß- und Kleinschreibung in der Direktive und im Programmcode exakt gleich sein muss.
- Der Name dieser Klasse wird zum Namen des Webservices, den der Proxyklassenassistent als Namen für die Proxyklasse verwendet.
- Die Klasse sollte durch die Annotation `[System.Web.Services.WebService]` Metadaten erhalten (Namensraum, Beschreibung und einen Webservicesnamen).
- Die Klasse sollte durch die neue Annotation `[WebServiceBinding(ConformsTo=WSIProfiles.BasicProfile1_1, EmitConformanceClaims=True)]` die Konformität zum WS-I Basic Profile ausdrücken.
- Jede Methode der Klasse, die sie als Webmethode im Rahmen des Webservice zur Verfügung stellen soll, muss mit der Annotation `<System.Web.Services.WebMethod>` markiert sein.
- Die Methode darf nur Parameter und einen Rückgabewert eines Typs besitzen, der serialisierbar ist. Die serialisierbaren Typen dürfen wiederum nur serialisierbare Datenmitglieder besitzen. Über einen XML-Webservice kann man also nicht nur elementare Datentypen wie `Integer`, `String` und `Boolean`, sondern auch komplexe Datentypen austauschen.
- Die Klasse sollte von `System.Web.Services.WebService` (oder einer Klasse, die davon erbt) erben, muss dies aber nicht tun. Die Verwendung dieser Klasse als Oberklasse hat jedoch den Vorteil, auf einfache Weise auf die eingebauten Objekte von ASP.NET zugreifen zu können.
- Webmethoden dürfen nicht überladen werden. Dies würde zu folgender Fehlermeldung führen: »x1 und x2 verwenden den Meldungsnamen x. Verwenden Sie die `MessageName`-Eigenschaft des benutzerdefinierten `WebMethod`-Attributs, um für Methoden eindeutige Namen anzugeben.«

Beispiel

Das folgende Beispiel zeigt die Implementierung des World Wide Wings Flugplan-Webservice. Die Klasse `WWWingsFlugplanService` ist als Webservice annotiert und besitzt vier als Webmethode annotierte Methoden. Dabei werden unterschiedliche Datentypen als Rückgabewerte dargestellt: der elementare Datentyp `long`, der komplexe in .NET eingebaute Typ `DataSet` und die Klasse `Flug` als selbstimplementierter komplexer Datentyp. Die vierte Methode `UpdateFlights()` erwartet als Parameter ein `DataSet`-Objekt und liefert keinen Rückgabewert. Webmethoden können Werte auch durch Referenzparameter an den Client zurückliefern.

HINWEIS

Typischerweise (so auch in diesem Beispiel) leiten die Webmethoden den Aufruf an die entsprechende Klasse der Geschäftslogik oder der Datenzugriffsschicht weiter.

Die Nutzung von ADO.NET-Typen schränkt die Interoperabilität zu anderen Plattformen ein, da dort Klassen wie `DataSet` und `DataTable` nicht zur Verfügung stehen.

```

<%@ WebService Language="c#" Class="WWWingsFlugplanService" %>
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;
using System;

[WebService(Namespace="http://worldwidewings.de/", Name="WorldWideWings Flugplan-Webservice",
Description="Webservices für den WorldWideWings-Flugplan!"),
WebServiceBinding(ConformsTo=wsisProfiles.BasicProfile1_1, EmitConformanceClaims=true)]
public class WWWingsFlugplanService : System.Web.Services.WebService
{

    [WebMethod()]
    public System.Data.DataSet GetAllFlights()
    {
        //Return New Data.DataSet()
        return de.WWWings.DAL.Flug_DataManager.AlleFluege_DS();
    }

    [WebMethod()]
    public long GetSeats(long FlightNo)
    {
        return Convert.ToInt64(new
de.WWWings.DAL.Flug_DataManager().GetFlug_DR(FlightNo)["Fl_Plaetze"].ToString());
    }

    [WebMethod()]
    public de.WWWings.Flug GetFlight(long FlightNo)
    {

        return de.WWWings.FlugBLManager.HoleFlug(FlightNo);
    }

    [WebMethod()]
    public void UpdateFlights(System.Data.DataSet ds)
    {
        de.WWWings.DAL.Flug_DataManager.Update(ds);
    }

    [WebMethod(EnableSession=true)]
    public void Zaehler(ref long BenutzerAufrufe, ref long Gesamtaufrufe)
    {
        Session["Counter"] = (long) Session["Counter"]+ 1;
        Application["Counter"] = (long) Application["Counter"]+ 1;
        BenutzerAufrufe = (long) Session["Counter"];
        Gesamtaufrufe = (long) Application["Counter"];
    }

    [WebMethod(EnableSession=true)]
    public de.WWWings.MitarbeiterSystem.Mitarbeiterzuordnung<de.WWWings.MitarbeiterSystem.Pilot,
de.WWWings.MitarbeiterSystem.Pilot> PilotCoPilotZuordnung(de.WWWings.MitarbeiterSystem.Pilot Pilot,
de.WWWings.MitarbeiterSystem.Pilot Copilot)
    {

        de.WWWings.MitarbeiterSystem.Mitarbeiterzuordnung<de.WWWings.MitarbeiterSystem.Pilot,
de.WWWings.MitarbeiterSystem.Pilot> mz = new
de.WWWings.MitarbeiterSystem.Mitarbeiterzuordnung<de.WWWings.MitarbeiterSystem.Pilot,
de.WWWings.MitarbeiterSystem.Pilot>(Pilot, Copilot);
        return mz;
    }
}

```

Listing 28.1 Implementierung eines Webservice mit vier Webmethoden [/Webservice/WWWingsFlugplanService.asmx]

Erstellung einer WSDL-Beschreibung

Ein WSDL-Dokument, das von Clients zum Erzeugen eines Proxys erzeugt wird, muss man bei ASMX-Webservices nicht manuell erstellen. ASP.NET generiert automatisch ein WSDL-Dokument, wenn an die ASMX-Datei beim Aufruf der Parameter `?WSDL` angehängt wird:

```
http://localhost:8888/Webservice/WWingsFlugplanService.asmx?WSDL
```

```
- <s:element name="GetFlight">
- <s:complexType>
- <s:sequence>
  <s:element minOccurs="1" maxOccurs="1" name="FlightNo" type="s:long" />
</s:sequence>
</s:complexType>
</s:element>
- <s:element name="GetFlightResponse">
- <s:complexType>
- <s:sequence>
  <s:element minOccurs="0" maxOccurs="1" name="GetFlightResult" type="tns:Flug" />
</s:sequence>
</s:complexType>
</s:element>
- <s:complexType name="Flug">
- <s:complexContent mixed="false">
- <s:extension base="tns:MarshalByRefObject">
- <s:sequence>
  <s:element minOccurs="1" maxOccurs="1" name="Datum" type="s:dateTime" />
  <s:element minOccurs="1" maxOccurs="1" name="FlugNr" type="s:long" />
  <s:element minOccurs="0" maxOccurs="1" name="AbflugOrt" type="s:string" />
  <s:element minOccurs="0" maxOccurs="1" name="ZielOrt" type="s:string" />
  <s:element minOccurs="1" maxOccurs="1" name="Plaetze" type="s:long" />
  <s:element minOccurs="1" maxOccurs="1" name="FreiePlaetze" type="s:long" />
  <s:element minOccurs="1" maxOccurs="1" name="Nichtraucherflug" type="s:boolean" />
</s:sequence>
</s:extension>
</s:complexContent>
</s:complexType>
```

Abbildung 28.2 Ausschnitt aus dem generierten WSDL-Dokument für den WWings-Webservice

Testanwendung für Webservices

Das ASP.NET Page Framework erzeugt automatisch eine Testwebsite für den Webservice, wenn die Seite nicht per SOAP, sondern per Webbrowser angesprochen wird. Die Testwebsite listet die enthaltenen Webmethoden auf und bietet zu jeder Webmethode einen Link zu einem automatisch generierten Testdokument. Dieses Testdokument bietet für jeden einfachen Parameterdatentyp ein Textfeld und eine Schaltfläche, um den Webservice aufzurufen. Die Testseite wird generiert, wenn die ASMX-Seite den Parameter *op* mit dem Namen der Webmethode erhält, also z.B.

`http://localhost:8888/Webservice/WWWingsFlugplanService.asmx?op=GetFlight.`

Das Ergebnis der Ausführung der Methode wird als einfaches XML-Dokument im Browser angezeigt.

HINWEIS Das Testformular funktioniert auf der Eingabeseite nur für einfache Datentypen und Arrays einfacher Datentypen. Bei Webmethoden mit komplexen Parametern kann das Testverfahren leider nicht eingesetzt werden. Der Rückgabewert darf beliebig komplex sein. Standardmäßig ist das Testformular aus Sicherheitsgründen nur auf dem lokalen System erreichbar.

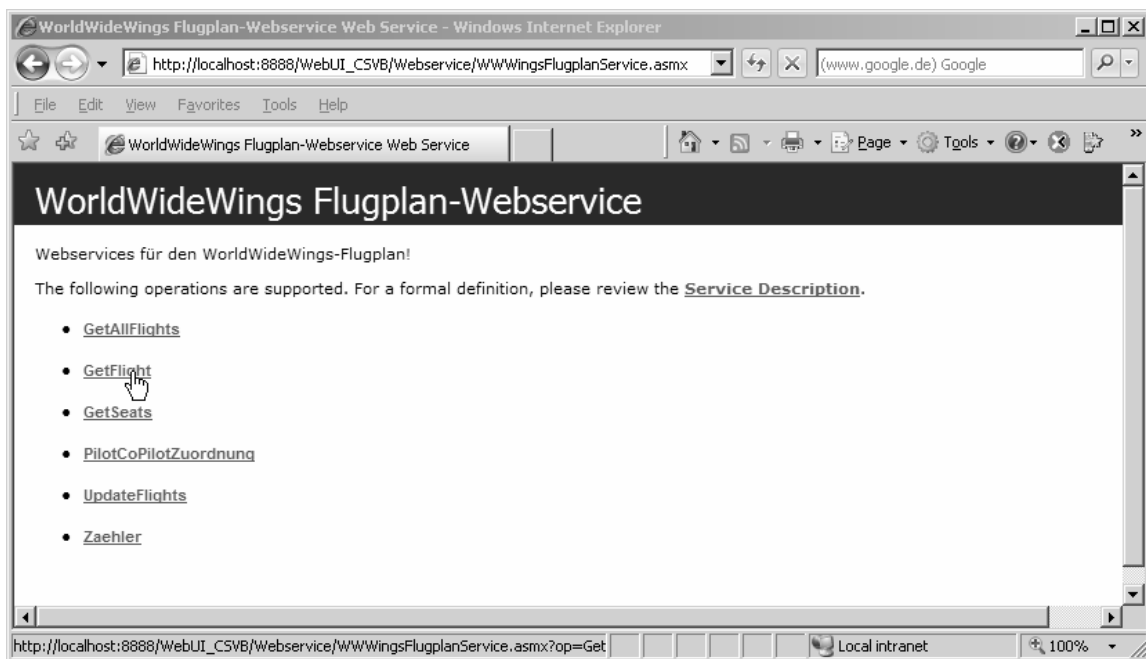


Abbildung 28.3 Testseite für einen ASP.NET-Webservice

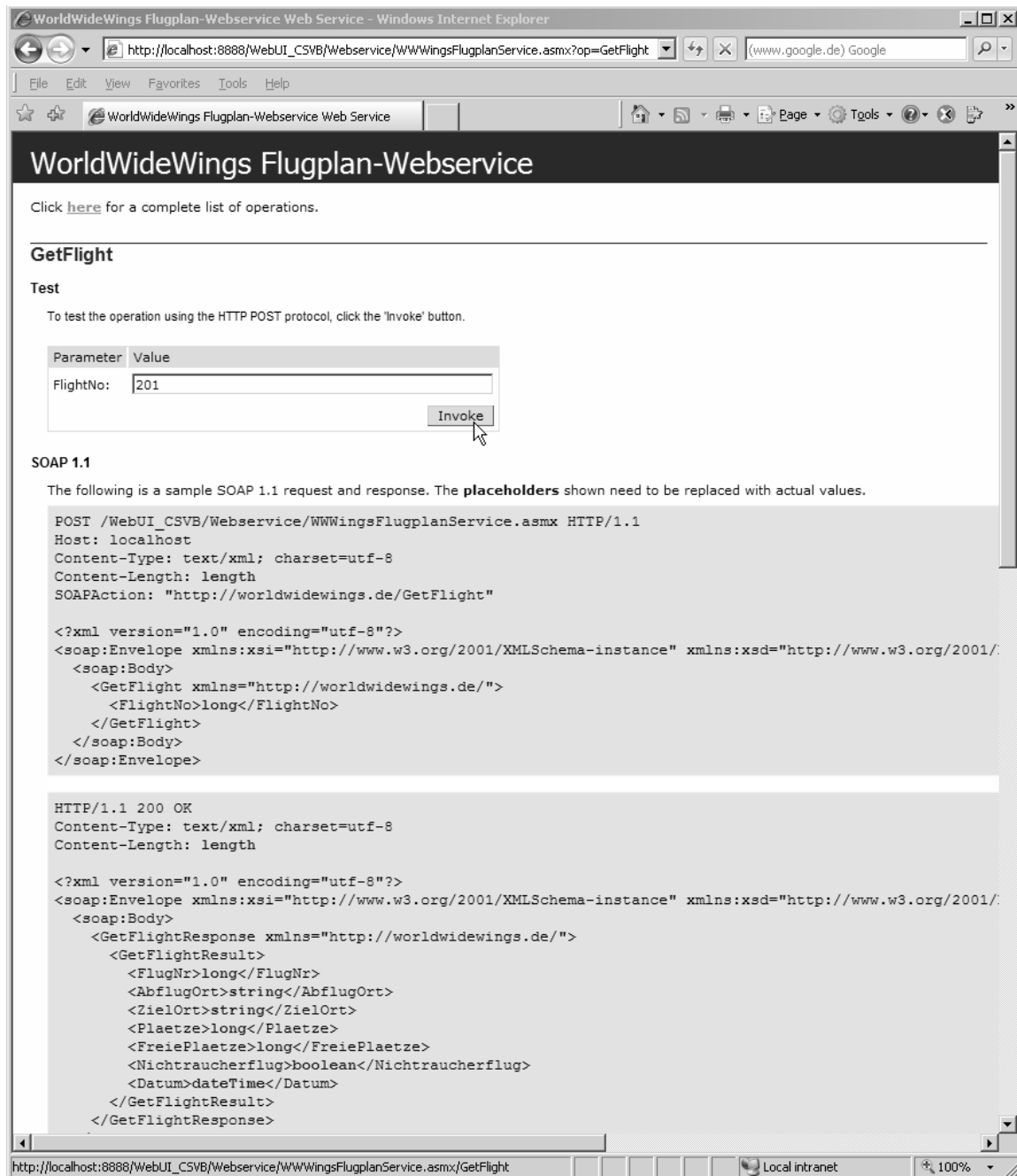


Abbildung 28.4 Testformular zum Aufruf einer Webmethode

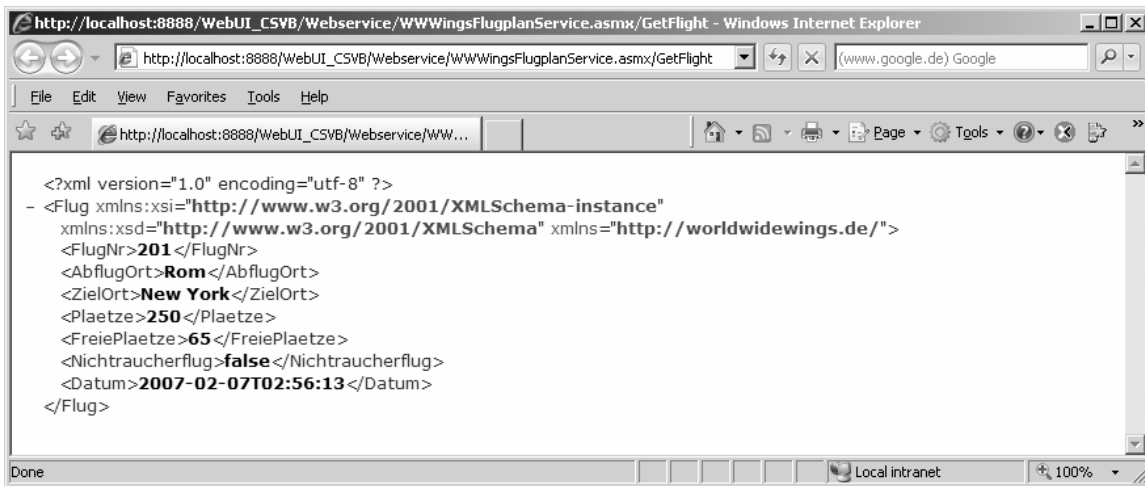


Abbildung 28.5 Ergebnis des Aufrufs der Webmethode

HINWEIS Wenn die Dokumentationsseite einen umfangreichen Hinweis auf <http://tempuri.org> (Temporary Unified Resource Identifier) enthält, haben Sie in der Annotation [WebService] keinen sinnvollen Namensraum angegeben. Seiten zum Aufruf sind nur für solche Methoden vorhanden, die einfache Datentypen als Parameter haben und deren Parameter mit *Call by Value* übergeben werden. Ein .NET-basierter Webservice warnt vor sich selbst auf der HTML-Testseite, wenn er nicht WS-I BP 1.0 konform ist.

TIPP Sie können die Webservicetestanwendung anpassen, indem Sie die Datei *DefaultWsdHelpGenerator.aspx* im Verzeichnis *%Framework%/Version%/Config* verändern.

Steuerung der SOAP-Serialisierung

Wie eingangs des Kapitels geschildert, unterstützt ASMX 2.0 alle vier SOAP-Arten. Die Dokumentation [MSDN17] behauptet jedoch, dass RPC/Literal keine Unterstützung erfährt. Die folgende Tabelle zeigt die notwendigen Annotationen. Standard ist die Kombination Document/Literal. Diese Kombination gilt, wenn keine der in der Tabelle angegebenen Annotationen vorhanden sind.

Die Serialisierung der in Parametern oder Rückgabewert übergebenen Objekte erledigt die Klasse `System.Xml.Serialization.XmlSerializer` bzw. `System.Runtime.Serialization.FormatterServices.SoapFormatter`, die man auch für eigene Zwecke nutzen kann. Die Serialisierer bieten ihrerseits wiederum eine Feinsteuerung der Serialisierung durch Meta-Attribute an, die direkt auf die Klassenmitglieder selbstdefinierter Klassen angewendet werden müssen. Dies sind z. B. `[XmlAttribute()]`, `[XmlIgnore()]`, `<SoapAttribute()>` und `[SoapIgnore()]`.

	Grundformat	
Parameterformat	Document	RPC
Literal	[SoapDocumentService(Use:=SoapBindingUse.Literal)] und [SoapDocumentMethod(Use:=SoapBindingUse.Literal)]	[SoapRpcService(Use:=System.Web.Services.Description.S SoapBindingUse.Literal)] und [SoapRpcMethod(Use:=System.Web.Services.Description.S SoapBindingUse.Literal)]
Encoded	[SoapDocumentService(Use:=SoapBindingUse.Encoded)] und [SoapDocumentMethod(Use:=SoapBindingUse.Encoded)]	[SoapRpcService(Use:=System.Web.Services.Description.S SoapBindingUse.Encoded)] und [SoapRpcMethod(Use:=System.Web.Services.Description.S SoapBindingUse.Encoded)]

Tabelle 28.1 Steuerung der SOAP-Formate durch Annotationen

Tipp zur Fehlerdiagnose

Wenn Sie genau sehen wollen, welche Daten Ihr Webserviceclient und Ihr Webservice austauschen, setzen Sie einen HTTP-Sniffer (z.B. Fiddler [FID01] oder MsSoapT aus dem Microsoft SOAP Toolkit) ein. Ein HTTP-Sniffer überwacht alle HTTP-Anfragen auf einem bestimmten Port, gibt sie aus und leitet sie dann zu einem anderen Port um.

Webservices in .NET-Anwendungen nutzen

Zur Implementierung eines Webserviceclients, der als Transportprotokoll HTTP nutzt, wird die Klasse `System.Web.Services.Protocols.SoapHttpClientProtocol` bereitgestellt. Diese Klasse erbt von der abstrakten Klasse `System.Web.Services.Protocols.HttpWebClientProtocol`, die wiederum von `System.Web.Services.Protocols.WebClientProtocol` erbt.

Die beiden wichtigsten Mitglieder der Klasse `SoapHttpClientProtocol` sind `Url` und `Invoke()`. `Url` legt die Adresse des aufzurufenden Webservice fest. `Invoke()` führt den Aufruf einer Webmethode durch und erwartet zwei Parameter: Im ersten Parameter wird der Name der aufzurufenden Webmethode und im zweiten Parameter ein Array vom Typ `Object` für die einzelnen Parameter der aufzurufenden Webmethode erwartet. Der Rückgabewert von `Invoke()` ist auch ein Array vom Typ `Object`. Hier werden nicht nur der eigentliche Rückgabewert der Webmethode, sondern auch alle Werte der Parameter zurückgeliefert, die als *Call by Reference* deklariert wurden.

Man kann allerdings `SoapHttpClientProtocol.Invoke()` nicht direkt aufrufen, denn `Invoke()` ist als `Protected` markiert. Es ist also notwendig, eine eigene Klasse zu implementieren und diese von `SoapHttpClientProtocol` abzuleiten. Diese Klasse wird als Clientproxy-Klasse bezeichnet.

Generierung der Proxyklasse

Microsoft stellt eine Hilfe zur Erzeugung der für einen Webservice notwendigen Proxyklasse(n) sowohl im Rahmen von Visual Studio als auch in Form des Befehlszeilenwerkzeugs *wsdl.exe* bereit.

Kommandozeile

Das Tool *wsdl.exe* ist Teil des .NET SDK. Hinter */language:* können als Sprachkürzel CS (für C#), VB (für Visual Basic) oder JS (für JScript .NET) angegeben werden. Die Ausgabedatei wird mit */out:* benannt. Als dritter Parameter ist der Pfad zu einem WSDL-Dokument zu spezifizieren:

```
wsdl /language:CS /out:WWingsServices.cs  
http://www.IT-Visions.de/WorldWideWings/Webservices/ WWingsFlugplanService.asmx
```

Visual Studio 2008

In Visual Studio 2008 gibt es zwei Möglichkeiten, einen Proxy zu erstellen:

- *Webverweis hinzufügen (Add Web Reference)*: erstellt einen Proxy auf Basis der Klasse `System.Web.Services.Protocols.SoapHttpClientProtocol`, die es schon in .NET 1.0 gab. Dies ist ein Proxy im Stil von ASMX.
- *Dienstverweis hinzufügen (Add Service Reference)*: erstellt einen Proxy auf Basis der Klasse `System.ServiceModel.ClientBase`, die es schon in .NET 1.0 gab. Dies ist ein Proxy im Stil der Windows Communication Foundation (WCF).

An dieser Stelle wird aus zwei Gründen nur die erste Möglichkeit besprochen: Zum einen läuft die zweite Möglichkeit nur auf Betriebssystemen ab Windows XP, und zum anderen ist die WCF kein Thema in diesem Buch (die WCF finden Sie in dem Buch »*.NET 3.5 Crashkurs*«).

ACHTUNG *Webverweis hinzufügen (Add Web Reference)* erscheint in einigen Projekttypen nur im Kontextmenü eines Projekts und im Menü *Project*, wenn als *Target Framework* das *.NET Framework 2.0* gewählt ist. Falls Sie den Eintrag nicht finden, müssen Sie etwas umständlich *Dienstverweis hinzufügen (Add Service Reference)* aufrufen, dann *Erweitert (Advanced)* anklicken und dort *Webverweis hinzufügen (Add Web Reference)* wählen.

Das Fenster *Webverweis hinzufügen (Add Web Reference)* bietet neben einer Eingabezeile für einen URL zu einem WSDL-Dokument auch Links zum Microsoft UDDI-Verzeichnis. Nach der Eingabe eines URL zeigt der Assistent die in dem WSDL-Dokument beschriebenen Operationen an und bietet die Schaltfläche *Verweis hinzufügen (Add Reference)* an. Nach dem Hinzufügen des Verweises erscheint dieser in dem Ast *Webverweise* im Projektmappen-Explorer.

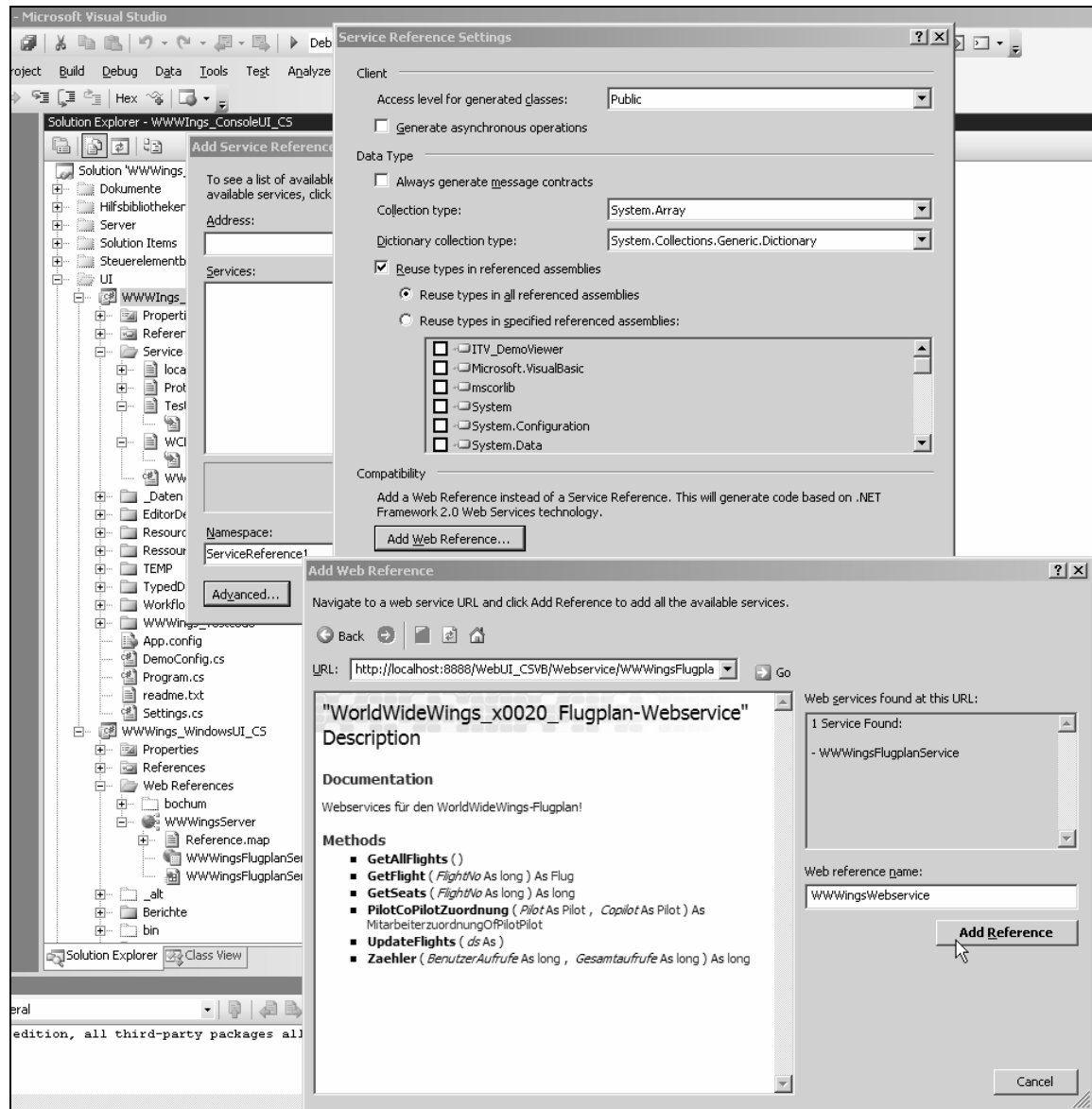


Abbildung 28.6 Gut versteckt: Das Erstellen eines Webserviceproxys im ASMX-Stil

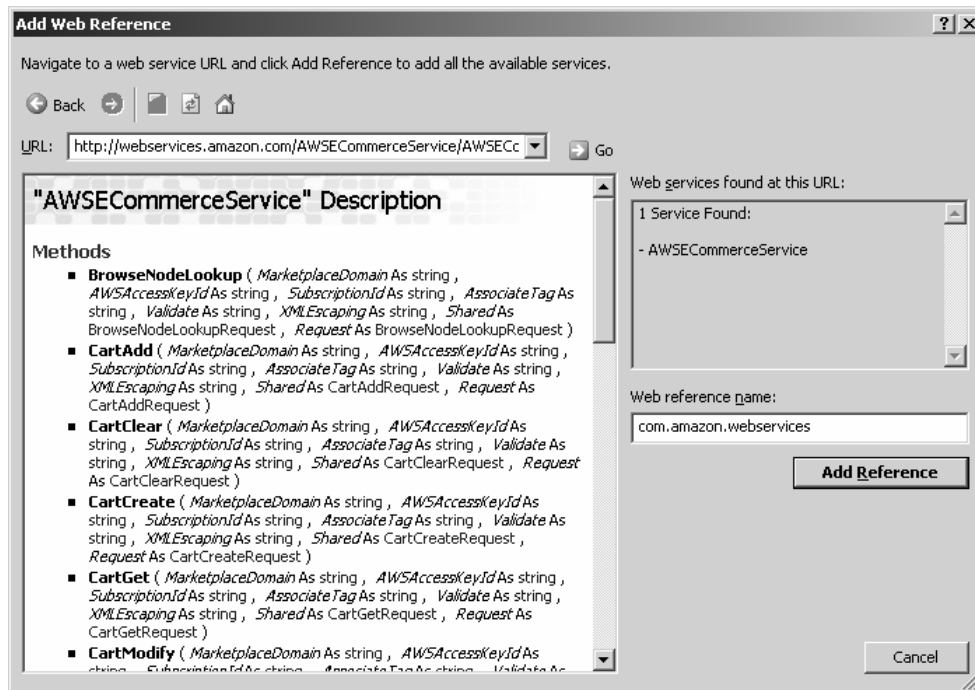


Abbildung 28.7 Erstellen eines Clientproxys für den (nicht in .NET implementierten) Amazon-E-Commerce-Webservice

Visual Studio erzeugt eine lokale Kopie des WSDL-Dokuments. Die generierte Codedatei (*Reference.vb* bzw. *Reference.cs*) mit der Proxyklasse sieht man nur, wenn die Funktion *Alle Dateien anzeigen* (*Show All Files*) im Projektmappen-Explorer aktiviert ist. Beim Anlegen eines Webserviceverweises ist ein Name anzugeben. Dieser wird nicht nur als Name für einen Unterordner, sondern auch als Namensraum für die Proxyklasse verwendet.

HINWEIS Visual Studio und *wsdl.exe* warnen, wenn Sie einen Webservice nutzen wollen, der nicht konform zu WS-I BP 1.0 ist.

In den Proxygeneratoren gibt es seit .NET-Version 2.0 folgende Neuerungen:

- Generische Typen können in Parametern und Rückgabewerten in Webservices verwendet werden. In der Proxyklasse erscheinen sie jedoch als nicht generische Typen.
- Wertelose Wertetypen werden unterstützt (dies ist die Standardeinstellung).
- Die Proxyklassen erhalten Getter- und Setter-Methoden (*Properties*) statt einfacher Datenmitglieder (*Fields*). Bei der Nutzung von *wsdl.exe* kann mit der Option */fields* das alte Verhalten erzwungen werden.
- Die Generierung ist über die Klasse *SchemaImporterExtension* komplett konfigurierbar.

HINWEIS Visual Studio bemerkt Änderungen an einem Webservice nicht automatisch. Wenn sich die Schnittstelle eines Webservice geändert hat (wenn beispielsweise eine neue Webmethode hinzugekommen ist), dann muss die Proxyklasse neu generiert werden. Dies geschieht durch den Eintrag *Webverweis aktualisieren (Update Web Reference)* im Kontextmenü eines Webverweises im Visual Studio-Projektmappen-Explorer.

Aufruf der Proxyklasse

Wenn man eine Proxyklasse erzeugt hat, kann man den Webservice so ansprechen, als wäre er eine lokale Klasse:

```
WWWingsServer.WorldWideWingsFlugplanWebservice ws = new
WindowsUI.WWWingsServer.WorldWideWingsFlugplanWebservice();
WWWingsServer.Flug f = ws.GetFlight(102);
Console.WriteLine("{0} fliegt von {1} nach {2}.", f.FlugNr, f.AbflugOrt, f.ZielOrt);
```

Listing 28.2 Aufruf einer Webmethode [/WindowsUI/WebserviceClient/WebserviceClient_EinfacheAusgaben]

Visual Studio bietet für die Proxyklasse Eingabeunterstützung per IntelliSense an. Bei der Nutzung von IntelliSense fällt auf, dass die Proxyklasse `WorldWideWingsFlugplanWebservice` mehr Mitglieder als nur die von dem Webservice bereitgestellten Methoden `GetAllFlights()`, `GetFlug_DR(FlightNo)`, `GetFlight(ByVal FlightNo As Long)` und `UpdateFlights(ByVal ds As System.Data.DataSet)` anbietet. Die übrigen stellen die Attribute und Methoden der Basisklasse `SoapHttpClientProtocol` dar.

Drei der zusätzlichen Attribute sind von besonderem Interesse:

- `Url` bietet die Möglichkeit, vor dem Aufruf der Methode den URL des Webservice zu setzen. Der `Webserviceclient` kann also den anzusprechenden Server zur Laufzeit auswählen, wenn mehrere Server den gleichen Dienst anbieten.
- Mit `Timeout` kann festgelegt werden, wie viele Millisekunden auf eine Antwort des Webservice gewartet werden soll. Kommt die Antwort nicht rechtzeitig, wird eine Ausnahme (*WebException: The operation has timed-out.*) ausgelöst.
- Im Attribut `Proxy` kann ein Objekt vom Typ `System.NET.WebProxy` übergeben werden, wenn für den Zugriff auf den Webservice ein Proxyserver zu überwinden ist.

Im folgenden Beispiel wird der URL explizit angegeben, ein Proxyserver gesetzt und dem Webservice 1.000 Millisekunden Zeit zum Antworten gegeben:

```
WWWingsServer.WorldWideWingsFlugplanWebservice ws2 = new
WindowsUI.WWWingsServer.WorldWideWingsFlugplanWebservice();
ws2.Url = "http://localhost:15115/Webservice/WWWingsFlugplanService.asmx";
ws2.Timeout = 1000;
ws2.Proxy = new System.NET.WebProxy("192.168.123.254", 80);
WWWingsServer.Flug f2 = ws2.GetFlight(102);
Demo.Print("{0} fliegt von {1} nach {2}.", f2.FlugNr, f2.AbflugOrt, f2.ZielOrt);
```

Listing 28.3 Aufruf einer Webmethode [/WindowsUI/WebserviceClient/WebserviceClient_EinfacheAusgaben]

Gemeinsame Datentypen (Proxy Type Sharing)

Normalerweise erzeugen die Proxygeneratoren für jeden Webservice – also für jede *.asmx*-URL – eigene Proxyklassen. Es kann aber vorkommen, dass *Webservice1.asmx* und *Webservice2.asmx* als Parameter die gleiche Klasse *y* verwenden. Daraus würden auf dem Client zwei Klassen *y1* und *y2* entstehen, die untereinander nicht mehr kompatibel wären. Ab .NET 2.0 bietet das Werkzeug *wsdl.exe* daher eine zusätzliche Option, um für mehrere *.asmx*-URLs gemeinsamen Proxycode zu erzeugen, sofern die Schemabeschreibungen in den WSDL-Dokumenten identisch sind:

```
wsdl.exe /out:ServiceProxies.cs /shareTypes URL1 URL2 URL3 ...
```

Asynchroner Aufruf

Bereits die vorherigen .NET-Versionen haben den asynchronen Aufruf von Webservices unterstützt. In .NET 1.x basierte der asynchrone Aufruf auf dem *IAAsyncResult*-Muster: Zu jeder Webmethode wurde in der Proxyklasse ein zusätzliches Methodenpaar *BeginMethodenname()* / *EndMethodenname()* generiert. In ASMX 2.0 wurde das Verfahren auf das Ereignismodell geändert: Nun wird zu jeder Webmethode eine zusätzliche Methode *MethodennameAsync()* sowie ein Ereignis *MethodennameCompleted()* generiert.

Ein asynchroner Aufruf kann durch die Methode *CancelAsync()* abgebrochen werden. Auch in diesem Fall erfolgt ein Aufruf der Ereignisbehandlungsroutine, allerdings mit dem Parameter *args.Cancelled = true*. Das folgende Beispiel zeigt den asynchronen Aufruf der Webmethode *GetFlight()*:

```
public void AsynchronerAufruf()
{
    WWWingsServer.WorldWideWingsFlugplanWebservice ws3 = new
        WindowsUI.WWWingsServer.WorldWideWingsFlugplanWebservice();
    ws3.GetFlightAsync(102);
    ws3.GetFlightCompleted += new
        WindowsUI.WWWingsServer.GetFlightCompletedEventHandler(ws3_GetFlightCompleted);
}

void ws3_GetFlightCompleted(object sender, WindowsUI.WWWingsServer.GetFlightCompletedEventArgs args)
{
    if (!args.Cancelled)
    {
        WWWingsServer.Flug f = args.Result;
        Demo.Print("Asynchrones Ergebnis: {0} fliegt von {1} nach {2}.", f.FlugNr, f.AbflugOrt, f.ZielOrt);
    }
    else
    {
        Demo.Print("Asynchroner Aufruf wurde abgebrochen.");
    }
}
```

Listing 28.4 Asynchroner Aufruf einer Webmethode [/WindowsUI/WebserviceClient/WebserviceClient_EinfacheAusgaben]

HINWEIS Die Änderung des asynchronen Aufrufs führt nicht zu Migrationsproblemen, da die Änderungen nicht die Basisklasse `SoapHttpClientProtocol` betreffen, sondern nur den generierten Proxyklassencode. Alte Proxyklassen können auch nach dem Import in Visual Studio (ab Version 2005) noch auf die gleiche Weise wie vorher genutzt werden.

Die Klasse `SoapHttpClientProtocol` stellt neben der `Invoke()`-Methode auch noch die drei Methoden `BeginInvoke()`, `EndInvoke()` und `InvokeAsync()` bereit. Auf dieser Basis generiert der Proxyklassenassistent für jede Webmethode die asynchronen Aufrufmethoden.

Authentifizierung

Ein Webserver kann einen Webservice durch Authentifizierung so schützen, dass nur bestimmte Benutzer ihn überhaupt aufrufen können. Bei der Verwendung der Microsoft IIS kann man im Internetdienstemanager zwischen den Authentifizierungsmethoden *anonym* (keine Authentifizierung), *Windows* (alias NTLM-Authentifizierung), *Standard* (Kennwörter werden im Klartext übertragen), *Digest* und *.NET Passport* wählen.

Der Webserviceproxy übermittelt in einer Standardeinstellung keine Authentifizierungsinformationen an den Webserver. Sofern der Webserver diese fordert, sind diese explizit anzugeben. Dabei besteht die Wahl zwischen der Weiterleitung der aktuellen Windows-Identität, unter welcher der Proxycode läuft (bei Windows Forms-Anwendungen also der angemeldete Benutzer), oder man kann ein anderes Benutzerkonto dafür verwenden. Im letzten Fall ist es aus Sicherheitsgründen besser, einen so genannten `CredentialCache` zu verwenden, als direkt eine Instanz der Klasse `NetworkCredential` an die `Credentials`-Eigenschaft zu übergeben, denn bei `CredentialCache` hat man Einfluss darauf, welches Authentifizierungsverfahren man unterstützen möchte. Wenn man direkt `NetworkCredential` instanziiert und der Server nur Standardauthentifizierung unterstützt, geht das Kennwort unverschlüsselt über das Netzwerk.

WICHTIG Die Kennwörter sollten nicht als Klartext im Code stehen, sondern verschlüsselt in der Anwendungskonfigurationsdatei abgelegt sein.

TIPP `ws2.PreAuthenticate = true` sorgt dafür, dass die Verhandlung zwischen Client und Webserver über das Authentifizierungsverfahren abgekürzt wird. Bitte beachten Sie aber, dass dadurch aus Sicherheitsgründen immer Authentifizierungsinformationen vom Client an den Server übermittelt werden, was eigentlich aus Sicherheitsgründen erst auf explizite Anforderung des Servers geschieht.

```
// Weiterleiten des aktuell angemeldeten Benutzers
WWWingsServer.WorldWideWingsFlugplanWebservice ws1 = new
    WindowsUI.WWWingsServer.WorldWideWingsFlugplanWebservice();
ws1.Credentials = System.Net.CredentialCache.DefaultNetworkCredentials;
ws1.PreAuthenticate = true;

// Explizite Identität
WWWingsServer.WorldWideWingsFlugplanWebservice ws2 = new
    WindowsUI.WWWingsServer.WorldWideWingsFlugplanWebservice();
ws2.Credentials = new System.Net.NetworkCredential("hp", "geheim", "worldwidewings");
ws2.PreAuthenticate = true;
```



```
// Explizite Identität und expliziter Authentifizierungsmechanismus
WWWingsServer.WorldWideWingsFlugplanWebservice ws3 = new
    WindowsUI.WWWingsServer.WorldWideWingsFlugplanWebservice();
System.Net.CredentialCache cache = new System.Net.CredentialCache();
cache.Add(new Uri(ws3.Url), "NTLM", // Basic, Kerberos, NTLM oder Negotiate
    new System.Net.NetworkCredential("ar", "geheim", "IT0"));
ws3.Credentials = cache;
ws3.PreAuthenticate = true;
```

Listing 28.5 Authentifizierung des Webserviceclients gegenüber dem Webserver

Weitere Möglichkeiten

Dieser Abschnitt beschreibt einige weitergehende Möglichkeiten von .NET-basierten XML-Webservices.

Fehlerbehandlung

Fehlerbehandlung ist bei Fernaufrufen nicht immer trivial. In SOAP existiert die Möglichkeit der Fehlerübertragung an den Client (Element <soap:Fault>). Das .NET Framework zeigt dem Client von SOAP übertragene Fehler immer als eine Ausnahme vom Typ `System.Web.Services.Protocols.SoapException` an, egal zu welcher Klasse die Ausnahme gehörte, die auf dem Server des Webservice ausgelöst wurde. Im Sinne der Entkopplung des Clients vom Webservice steht die eigentliche Ausnahme nicht in dem Attribut `InnerException`. Daraus folgt, dass Sie bei der Verwendung eigener Fehlerklassen immer darauf achten müssen, dass das `Message`-Attribut des Fehlers alle notwendigen Informationen enthält.

Das folgende Codefragment zeigt den Aufruf der Webmethode `UpdateFlights()` unter Nutzung der Fehlerbehandlung. Einen Fehlerzustand kann man leicht herbeiführen, indem man den Windows-Client zweimal öffnet, nacheinander die gleiche Zeile ändert und dann in beiden Fenstern speichert. Der daraus entstehende Änderungskonflikt löst in der Klasse `DataAdapter` eine `DbConcurrencyException` aus, die an den Client weitergereicht wird, sofern sie nicht in der Geschäftslogik abgefangen wird.

```
try
{
    ws.UpdateFlights(aenderungen);
    ds.AcceptChanges();
}
catch (System.Web.Services.Protocols.SoapException ex)
{
    System.Windows.Forms.MessageBox.Show("Fehler: " + ex.Message, "Fehler beim Aufruf des Webservice");
}
```

Listing 28.6 Fehlerbehandlung beim Webserviceaufruf [/WindowsUI/WebserviceClient/Flugverwaltung_Webservice]

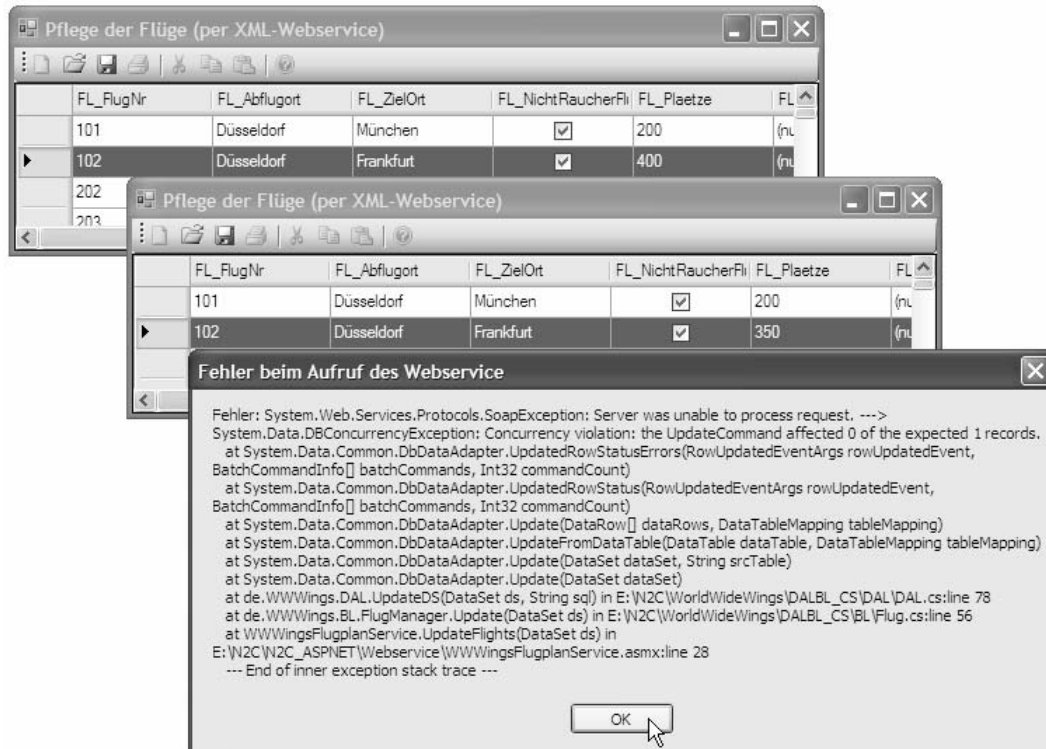


Abbildung 28.8 Verhalten des Webserviceclients bei einem Änderungskonflikt

Generierung der Serverschnittstellen

Neu in ASMX V2 ist die Möglichkeit, gemäß dem Contract First-Ansatz aus einem vorhandenen WSDL-Dokument die Schnittstellen zur Implementierung des Webservices zu generieren. Das Werkzeug *wsdl.exe* erzeugt bei Angabe eines Pfades zu einem WSDL-Dokument und der Option */serverinterface* (alias */si*) die Schnittstellendefinition für die ASMX-Klasse sowie eine partielle Klassendefinition für die beteiligten benutzerdefinierten Klassen. In dem obigen Beispiel des Flugplan-Webservices generiert das Werkzeug folgende Schnittstellen und Klassen:

- `public partial interface IWorldWideWingsFlugplanWebserviceSoap { ... }`
- `public partial class Flug { ... }`
- `public partial class Pilot : Mitarbeiter { ... }`
- `public partial class Mitarbeiter : Person { ... }`
- `public partial class Person { ... }`

Statische Generierung von Serialisierungsassemblies (sgen.exe)

XML-Webservices basieren auf der XML-Serialisierung aller beteiligten Daten. Das Standardverhalten von .NET ist, dass zur Laufzeit eine so genannte Serialisierungsassembly erzeugt wird, welche die eigentliche Serialisierung vornimmt. Bisher wurde diese Serialisierungsassembly nicht zur Entwicklungszeit, sondern beim ersten Aufruf erzeugt. Ab .NET 2.0 enthält das .NET SDK das Werkzeug *sgen.exe* (*XML Serialization Support Utility*), das eine Persistierung einer Serialisierungsassembly erlaubt. Diese zur Entwicklungszeit vom Entwickler des Webservice bereitgestellte Serialisierungsassembly kann vom Entwickler des Clients referenziert werden.

Voraussetzung für die Serialisierbarkeit eines benutzerdefinierten Typs ist, dass alle verwendeten Mitglieds-typen serialisierbar sind.

Der Aufruf von *sgen WWWings_GL.dll* erzeugt eine Assembly namens *WWWings_GL.XmlSerializers.dll* mit Implementierung der Serialisierung für alle serialisierbaren Typen in der Ausgangsassembly.

In Visual Studio können Sie die automatische Generierung der Serialisierungsassembly anstoßen, und zwar in den Projekteigenschaften, Registerkarte *Kompilieren* (*Build*), *Erweiterte Kompilierungsoptionen* (*Advanced Compile Options*), *Serialisierungsassemblies generieren* (*Generate serialization assemblies*).

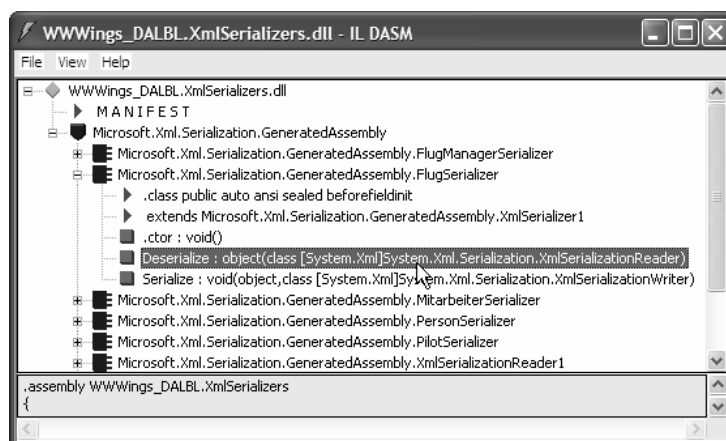


Abbildung 28.9 Ergebnis der automatischen Generierung der Serialisierungsassembly

ACHTUNG Vorteil der statischen Serialisierungsassembly ist die höhere Geschwindigkeit. Ein großer Nachteil liegt aber darin, dass die Serialisierungsassembly auf den Clients installiert und bei Änderungen der Klasse aktualisiert werden muss. Dadurch schwindet der Vorteil der losen Kopplung von XML-Webservices allein auf Basis eines WSDL-Dokuments.

Benutzerdefinierte Serialisierung

Die XML-Serialisierung eines Typs kann durch verschiedene Annotationen beeinflusst werden, z.B.:

- `System.Xml.Serialization.XmlAttributeAttribute` erreicht, dass das Attribut nicht als XML-Element, sondern als Attribut eines XML-Elements serialisiert wird.
- `System.Xml.Serialization.XmlIgnore` sorgt dafür, dass ein Attribut der Klasse nicht serialisiert wird.

Neu seit .NET Version 2.0 ist der vollständige Einfluss auf die XML-Serialisierung durch Implementierung der Schnittstelle `IXmlSerializable`. Dieses komplexe Thema kann hier jedoch aus Platzgründen nicht behandelt werden.

Zustandsbehaftete Webservices

Alle eingebauten Objekte von ASP.NET stehen auch innerhalb einer Webserviceklasse zur Verfügung – das gilt auch für die Klasse `Session`, in der benutzerbezogene Zustände gehalten werden können. ASP.NET-Webservices können daher zustandsbehaftet sein. Innerhalb einer Webserviceklasse können Werte in die `Session`-Objektmenge geschrieben und bei einem erneuten Aufruf desselben Benutzers wieder gelesen werden. Drei Punkte sind dabei zu beachten:

- Die Sitzungs-ID kann nur durch Cookies, nicht durch URL-Rewriting auf dem Client gespeichert werden, weil eine Webserviceanfrage nicht umgeleitet werden kann.
- Die Speicherung von Cookies ist eine Besonderheit eines Webbrowsers. Webservicesclients kennen normalerweise keine Cookies. Damit Sie Cookies aufnehmen können, brauchen Sie einen so genannten **Cookiecontainer**. Einen solchen stellt die FCL in der Klasse `System.NET.CookieContainer` bereit.
- Die Webmethode muss durch `<WebMethod(EnableSession:=True)>` gekennzeichnet werden.

Server

Das folgende Listing zeigt einen einfachen zustandsbehafteten Webservice, der zählt, wie oft er von einem bestimmten Benutzer, und wie oft er insgesamt aufgerufen wurde:

```
[WebMethod(EnableSession=true)]
public void Zaehler(ref long BenutzerAufrufe, ref long Gesamtaufrufe)
{
    Session["Counter"] = (long) Session["Counter"] + 1;
    Application["Counter"] = (long) Application["Counter"] + 1;
    BenutzerAufrufe = (long) Session["Counter"];
    Gesamtaufrufe = (long) Application["Counter"];
}
```

Listing 28.7 Eine zählende Webmethode [/WebUI/Webservice/WWWingsFlugplanService.asmx]

Windows Forms Client

Als erster Client soll hier eine Windows Forms-Anwendung zum Einsatz kommen. Wichtig ist dabei, dass der `Cookiecontainer` wiederverwendet, also nur beim ersten Aufruf instanziiert wird:

```
WWWingsServer.WorldWideWingsFlugplanWebservice ws = new
WindowsUI.WWWingsServer.WorldWideWingsFlugplanWebservice();
private void C_Aufruf_Click(object sender, EventArgs e)
{
    ws.Url = this.C_URL.Text;
    if (ws.CookieContainer == null) ws.CookieContainer = new System.NET.CookieContainer();
    long count1 = 0, count2 = 0;
```

```
ws.Zaehler(ref count1, ref count2);  
this.labell1.Text = count1.ToString();
```

Listing 28.8 Client für die zählende Webmethode [/WindowsUI/WebserviceClient/WebserviceClient_EinfacheAusgaben]

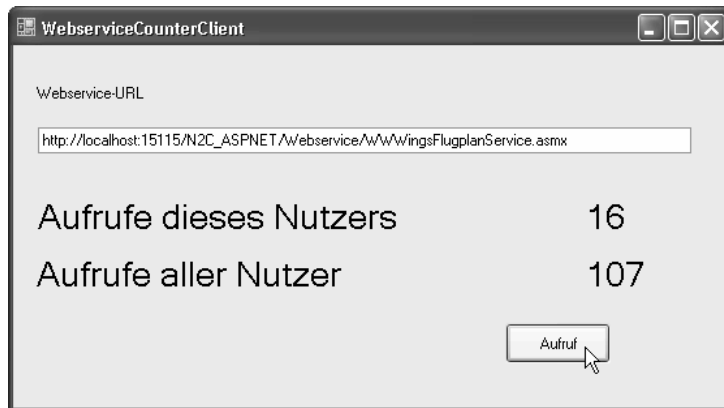


Abbildung 28.10 Client für die zählende Webmethode

Webservices nutzen in ASP.NET-Webanwendungen

Eine ASP.NET-Anwendung kann nicht nur Webservices bereitstellen, sondern auch selbst Webservices anbieten. Grundsätzlich kann man in einem Web Forms-Projekt eine Webreferenz erzeugen wie aus jedem anderen .NET-Projekttyp auch. Es gilt aber zwei Dinge zu beachten:

- Auf dem Windows-Betriebssystem ist die Anzahl der ausgehenden HTTP-Verbindungen jeweils auf zwei gleichzeitige Verbindungen pro Zielsystem begrenzt. Dies fällt weniger ins Gewicht, wenn ein Benutzer eine Windows Forms-Anwendung auf seinem Computer startet, weil er üblicherweise nicht zahlreiche Anfragen gleichzeitig zum selben Zielsystem sendet (Ausnahme: Er hat z.B. mehrere eBay-Clients gleichzeitig geöffnet). Bei einer Web Forms-Anwendung hingegen arbeiten immer unzählige Benutzer auf dem gleichen Webserver und dieser Webserver wird daher in der Regel mehr als zwei Webservice-Aufrufe gleichzeitig an den Anwendungsserver starten.
- Wenn der aufgerufene Webserver zustandsbehaftet ist und dafür Cookies verwendet, darf man nicht den Fehler machen, den Cookiecontainer bei der Implementierung als überflüssig zu erachten, weil ASP.NET doch Cookies unterstützt. Richtig ist: ASP.NET unterstützt das Senden von Cookies an den Browser, aber nicht die Ablage von Cookies, die der Programmcode in einem Web Form von einer anderen Webseite empfängt.

HINWEIS In Projekten nach dem Websitemodell landen alle Webproxys im Ordner *App_WebReferences*.

Aufhebung der HTTP-Begrenzung

Die HTTP-Begrenzung auf zwei gleichzeitige Verbindungen zum gleichen Zielsystem kann durch eine Einstellung in der *web.config*-Datei aufgehoben werden:

```
<system.net>
  <connectionManagement>
    <add address="*" maxconnection="999" />
  </connectionManagement>
</system.net>
```

Listing 28.9 Aufhebung der HTTP-Begrenzung bei Nutzung einer Web Forms-Anwendung als Webserviceclient

Implementierung eines Cookiecontainers

Das Web Form muss also einen Cookiecontainer bereitstellen, um die Sitzungs-ID, die es mit jeder HTTP-Antwort bekommt, speichern zu können. Diese muss in einer Sitzungsvariablen (*Page.Session*) abgelegt werden, weil der Cookiecontainer sonst bei jedem Neuaufwurf der Seite wieder leer ist.

Beispiel

In dem Beispiel wird ein Web Forms-Client für den zustandsbehafteten Webservice aus dem vorangegangenen Abschnitt realisiert. Dieses durch den Einsatz von zwei verschiedenen Benutzersitzungen nicht gerade triviale Szenario sei zusätzlich durch eine Grafik veranschaulicht (Abbildung 28.11).

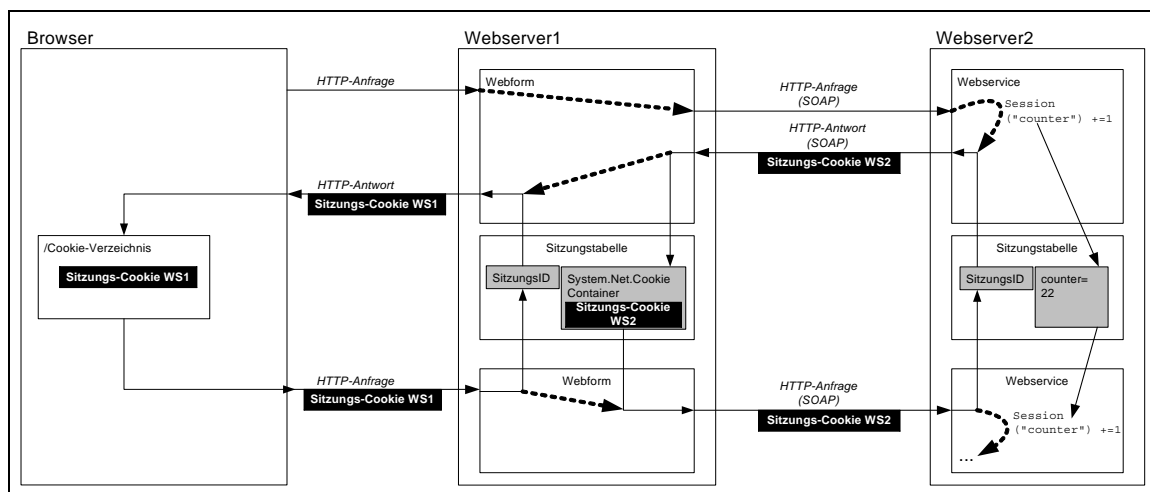


Abbildung 28.11 Ein Webform als Client für einen zustandsbehafteten Webservice



Abbildung 28.12 Webclient für den zustandsbehafteten Webservice

```
private void C_Aufruf_Click(object sender, System.EventArgs e)
{
    // --- Cookie Container bereitstellen
    if (Session["CC"] == null)
        Session["CC"] = new System.Net.CookieContainer();
    // --- Webservice instanziiieren
    ITVService.ITVDienste ws = new ITVService.ITVDienste();
    // --- Cookiecontainer zuweisen
    ws.CookieContainer = Session["cc"];
    // --- Speicher für Rückgabewerte bereitstellen
    long b = 0;
    long g = 0;
    // --- Webmethode aufrufen
    ws.Zaehler(b, g);
    // --- Ergebnisse ausgeben
    C_BCount.Text = b;
    C_GCount.Text = g;
}
```

Listing 28.10 Speichern eines Cookies im Webclient

DISCO-Unterstützung

ASMX unterstützt auch das DISCO (Discovery)-Protokoll zum Auffinden von Webservices. Dazu ist an den URL einer ASMX-Datei der Parameter *?disco* anzuhängen, z.B.

http://localhost:8888/WebUI_CSVB/Webservice/WWWingsFlugplanService.asmx?disco

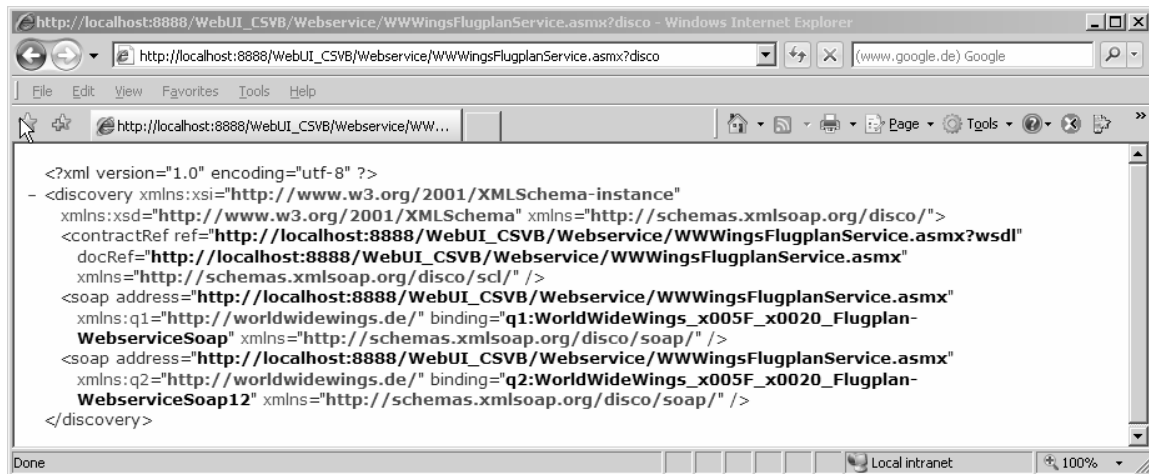


Abbildung 28.13 Anzeige eines DISCO-Dokuments

Web Services Enhancements (WSE) 3.0

Die *Web Services Enhancements* (WSE) sind eine Erweiterung für das .NET Framework, die zusätzliche Funktionen für XML-Webservices bereitstellen. Im Zuge von .NET 2.0 sind die WSE Version 3.0 mit folgenden Funktionen erschienen:

- Bereitstellen/Nutzen von Webservices direkt auf Basis von TCP (ohne HTTP) mit Adressierung per *soap.tcp://*,
- Authentifizieren von Webserviceaufrufen auf Nachrichtenebene (via Benutzername/Kennwort, Kerberos und X.509-Zertifikaten),
- Verschlüsseln von Webserviceaufrufen,
- Routen von Webserviceaufrufen,
- Nachrichtenoptimierung mit SOAP Message Transmission Optimization Mechanism (MTOM).

WSE 3.0 ist ein Add-On für .NET 2.0 (Namensraum `Microsoft.Web.Services3`), das separat installiert werden muss. Visual Studio erhält durch die Installation einen WSE-Assistenten. Unterstützung für die Vorgängerversion WSE 2.0 ist nur noch zur Laufzeit vorhanden. Visual Studio Versionen 2005 und 2008 bieten keine Hilfe mehr bei der Verwendung von WSE 2.0. Die Funktionen der WSE sind nun in der Windows Communication Foundation (WCF) enthalten.