2 WCF im Überblick

Dieses Kapitel erläutert zunächst die wichtigsten Grundlagen von WCF und zeigt dann anhand eines ersten Beispiels, wie damit SOAP-basierte Services umgesetzt und konsumiert werden können. Kapitel 3, „Service mit WCF erstellen“, geht anschließend auf Details der WCF ein.

2.1 Architektur

Mit WCF stellt Microsoft ein Framework mit einem einheitlichen Programmiermodell zur Implementierung von Services zur Verfügung. „Einheitliches Programmiermodell“ bedeutet an dieser Stelle, dass mit denselben Mitteln verschiedenartige Services implementiert werden können, darunter Services, die direkt auf TCP aufsetzen und ein effizientes binäres Datenformat verwenden, oder Services, die über HTTP und SOAP/XML angesprochen werden. Technische Details wie das zu verwendende Protokoll, die zu verwendenden Datenformate oder Aspekte in Hinblick auf Sicherheit werden über die Konfiguration festgelegt. Metadaten zu den konfigurierten Services können entweder via HTTP(S) als WSDL-Datei oder via HTTP(S), TCP und Named Pipes unter Verwendung von *WS-Metadata-Exchange* exportiert werden. Diese Metadaten sind die Grundlage für die Erstellung von Proxys.

Pro Service können beliebig viele Endpunkte definiert werden, wobei jeder Endpunkt eine Adresse aufweist und sich auf ein Binding sowie auf einen Service-Vertrag (*Contract*) bezieht. Als Akronym für diese drei Merkmale wird ABC (*Address*, *Binding*, *Contract*) herangezogen.

Bindings legen das zu verwendende Transportprotokoll (TCP, Named Pipes, HTTP, MSMQ) sowie das Nachrichtenformat (binär, XML, JSON, SOAP) fest. Darüber hinaus können sie sich auch auf weitere Protokolle beziehen, wie zum Beispiel Protokolle für Sicherheitsszenarien, verlässliche Zustellungen oder Transaktionen. Daneben besteht auch die Möglichkeit, benutzerdefinierte Bindings zu konfigurieren. Kommt man damit auch nicht aus, kann man über das Erweiterungsmodell von WCF eigene Bindings implementieren.

Der Service-Vertrag legt fest, welche Service-Operationen über den jeweiligen Endpunkt aufgerufen werden können. Technisch gesehen handelt es sich dabei um ein Interface, das

über Attribute Metadaten bereitstellt und vom Service zu implementieren ist, wobei jeder Service beliebig viele solcher Interfaces implementieren kann.

Daneben können für Endpunkte und Services sogenannte Verhalten (*Behaviors*) konfiguriert werden. Diese spiegeln Aspekte wider, die sich zwar auf das Laufzeitverhalten, nicht jedoch auf die angebotenen Operationen auswirken. In vielen Fällen sind die konfigurierten Verhalten für den Service-Konsumenten nicht direkt ersichtlich. Beispiele hierfür sind die Art der Prüfung der übersendeten Benutzerkennungen oder die Beschränkung gleichzeitiger Zugriffe sowie die Protokollierung von Nachrichten.

2.2 Standard-Bindings

Bevor die folgenden Kapitel des vorliegenden Buches näher auf die einzelnen Möglichkeiten der in WCF inkludierten Bindings, gruppiert nach Themengebieten, eingehen, werden sie in diesem Abschnitt im Überblick vorgestellt. Zu diesem Zweck bietet Tabelle 2.1 eine grobe Übersicht.

Tabelle 2.2 beschreibt die von den einzelnen Bindings bereitgestellten Funktionen, wobei das Standardverhalten jeweils in Klammern gesetzt wurde. Beide Tabellen wurden leicht verändert aus der MSDN-Dokumentation übernommen (tinyurl.com/3hyvbrs)

Bei Betrachtung von Tabelle 2.1 fällt auf, dass es anscheinend zwei Gruppen von Bindings gibt. Die HTTP-basierten Bindings verwenden offene Standards, wie zum Beispiel SOAP, und finden vor allem bei der Kommunikation zwischen unterschiedlichen Systemen Anwendung. Die „net"-Bindings verwenden hingegen effizientere binäre Mechanismen, die nicht interoperable sind und deswegen in erster Linie für die Kommunikation zwischen WCF-basierten Systemen oder Teilen eines WCF-basierten Systems herangezogen werden.

Die Spalte **Sicherheitsmodus** in Tabelle 2.2 gibt an, auf welcher Ebene die Kommunikation verschlüsselt werden kann. *Transport* weist hier auf eine Verschlüsselung auf Transportebene, zum Beispiel mittels HTTPS, hin. *Message* bedeutet hingegen, dass die Nachricht an sich verschlüsselt werden kann und somit kein sicheres Transportprotokoll mehr vonnöten ist.

Die Spalte *Sitzung* informiert darüber, ob der Service zwischen den einzelnen Aufrufen desselben Konsumenten Daten zwischenspeichern kann. Als Beispiel sei hier ein Service zur Implementierung eines Warenkorbes genannt. Werden Sitzungen unterstützt, kann der Konsument mit verschiedenen Aufrufen die gewünschten Produkte nach und nach in den durch den Service bereitgestellten Warenkorb legen und am Ende der Einkaufstour alle dort gesammelten Produkte gemeinsam bestellen. Werden keine Sitzungen unterstützt, müssen alle gewünschten Produkte gemeinsam im Zuge eines einzigen Aufrufs, der die Bestellung tätigt, übergeben werden. Der Wert Transport deutet darauf hin, dass Sitzungen über das verwendete Transportprotokoll implementiert werden. Dies setzt natürlich ein Transportprotokoll wie TCP voraus, welches das Konzept von Sitzungen kennt. *Zuverlässige Sitzung* informiert hingegen darüber, dass zur Schaffung von Sitzungen *WS-SecureConversation* eingesetzt wird. Diese Spezifikation sieht vor, dass jede Nachricht eine Sitzungs-ID beinhaltet, sodass der Service den aktuellen Aufruf einer Sitzung zuordnen kann. Somit muss das

darunter liegende Transportprotokoll das Konzept von Sitzungen auch nicht unterstützen. Ferner stellt *WS-SecureConversation* durch Nummerieren von Nachrichten sowie durch den Einsatz von Bestätigungsnachrichten sicher, dass alle gesendeten Nachrichten vom Service in der vorgesehenen Reihenfolge verarbeitet werden.

Die Spalte *Transaktionen* informiert darüber, ob das jeweilige Binding das Ausdehnen von Transaktionen über Servicegrenzen hinweg unterstützt, und *Duplex*, ob der Service von sich aus die Möglichkeit hat, Konsumenten über eingetretene Ereignisse zu informieren.

Tabelle 2.1 Standard-Bindings

Bindung	Beschreibung
BasicHttpBinding	Verwendet die ältere SOAP-Version 1.1 und unterstützt wenige WS-*-Protokolle. Kommt vor allem in Szenarien, wo auf Abwärtskompatibilität geachtet werden muss, zum Einsatz.
WSHttpBinding	Verwendet die aktuelle SOAP-Version 1.2 und bietet Unterstützung für WS-*-Spezifikationen. Ist nicht für Duplex-Szenarien geeignet.
WS2007HttpBinding	Wie WSHttpBinding, allerdings werden aktuellere Versionen der unterstützten WS-*-Spezifikationen herangezogen
WSDualHttpBinding	Wie WSHttpBinding, allerdings werden auch Duplex-Szenarien unterstützt
WSFederation-HttpBinding	Wie WSHttpBinding, allerdings wird WS-Federation unterstützt. Dies erlaubt die Etablierung von Vertrauensstellungen zwischen verschiedenen Sicherheitsdomänen und somit zum Beispiel einen unternehmensübergreifenden Einsatz von Benutzerkonten. Mehr Details hierzu finden Sie in Kapitel 4, „Sicherheit von WCF-Dienster“.
WS2007Federation-HttpBinding	Wie WSFederationHttpBinding, allerdings werden aktuellere Versionen der unterstützten WS-*-Spezifikationen herangezogen
NetHttpBinding	Verwendet ein binäres Protokoll über HTTP und erlaubt den Einsatz von WebSockets für Firewall-sichere Benachrichtigungs-Szenarien über HTTP.
NetTcpBinding	Verwendet ein binäres Protokoll und basiert direkt auf TCP. Wird zur effizienten Kommunikation zwischen WCF-basierten Systemen eingesetzt
NetNamedPipe-Binding	Verwendet, wie NetTcpBinding, ein effizientes binäres Protokoll. Wird zur Kommunikation zwischen Systemen auf demselben Rechner eingesetzt und basiert auf *Named Pipes*.
NetMsmqBinding	Verwendet Microsoft Message Queues sowie ein binäres Protokoll zur verlässlichen und asynchronen Kommunikation
NetPeerTcpBinding	Verwendet ein binäres Protokoll zur Peer-to-Peer-Kommunikation
WebHttpBinding	Wird für REST-Szenarien herangezogen
MsmqIntegration-Binding	Basiert, wie auch NetMsmqBinding, auf Microsoft Message Queues und bietet eine verlässliche und asynchrone Art der Kommunikation. Im Gegensatz zu NetMsmqBinding kann es zur Kommunikation mit Systemen, die nicht auf WCF basieren, eingesetzt werden.
UdpBinding	Verwendet ein binäres Protokoll sowie UDP. Erlaubt Multicasts. Die Kommunikation kommt mit möglichst wenig Protokoll-Overhead aus, ist dafür jedoch nicht zuverlässig, d. h. es können Informationen verloren gehen.

Tabelle 2.2 Standard-Bindings und deren Eigenschaften

Bindung	Sicherheitsmodus	Sitzung	Trans-aktionen	Duplex
BasicHttpBinding	(Keine), Transport, Nachricht, Gemischt	Keine (Keine)	(Keine)	nicht verfügbar
WSHttpBinding	(Keine), Transport, (Nachricht), Gemischt	(Keine), Transport, zuverlässige Sitzung	(Keine), Ja	nicht verfügbar
WS2007Http-Binding	(Keine), Transport, (Nachricht), Gemischt	(Keine), Transport, zuverlässige Sitzung	(Keine), Ja	nicht verfügbar
WSDualHttp-Binding	Keine, (Nachricht)	(Zuverlässige Sitzung)	(Keine), Ja	Ja
WSFederation-HttpBinding	Keine, (Nachricht), Gemischt	(Keine), zuverlässige Sitzung	(Keine), Ja	Nein
WS2007Federa-tionHttpBinding	Keine, (Nachricht), Gemischt	(Keine), zuverlässige Sitzung	(Keine), Ja	Nein
NetHttpBinding	(Keine), Transport, Nachricht, Gemischt	(Keine), zuverlässige Sitzung	(Keine), Ja	Ja
NetTcpBinding	Keine, (Transport), Nachricht, Gemischt	Zuverlässige Sit-zung, (Transport)	(Keine), Ja	Ja
NetNamed-PipeBinding	Keine, (Transport)	Keine, (Transport)	(Keine), Ja	Ja
NetMsmqBinding	Keine, Nachricht, (Transport), Beide	(Keine)	(Keine), Ja	Nein
NetPeerTcp-Binding	Keine, Nachricht, (Transport), Gemischt	(Keine)	(Keine)	Ja
MsmqIntegra-tionBinding	Keine, (Transport)	(Keine)	(Keine), Ja	nicht verfügbar Formularende
UdpBinding	Keine	(Keine)	(Keine)	nicht verfügbar

2.3 Hosting von Services

Zur Ausführung muss ein Service in einem Hostprozess gestartet werden. Dies kann eine benutzerdefinierte Applikation, die zum Beispiel als Windows-Service bereitgestellt wird, oder ein Web- bzw. Applikationsserver, wie die in Windows integrierten *Internet Information Services* (IIS), sein. Benutzerdefinierte Hosts unterstützen alle verfügbaren Bindings; IIS bis zur Version 6 allerdings nur solche, die auf HTTP(S) basieren. Seit IIS 7 ist diese Einschränkung dank der *Windows Activation Services* (WAS), die Protokoll-Listener für Named Pipes, TCP und MSMQ anbieten, beseitigt worden. Um in den Genuss dieser Technologie zu kommen, muss die Windows-Funktion mit der Bezeichnung *Windows-Prozessaktivierungsdienst* in der Systemsteuerung unter *Programme* aktiviert werden. *Windows Server AppFabric* setzt auf IIS und WAS auf und bringt Erweiterungen zur Verwaltung sowie zum

Überwachen von Services mit sich. Daneben können WCF-Services auch im Entwicklungswebserver von Visual Studio sowie mit einem generischen Host, der ebenfalls mit Visual Studio mitausgeliefert wird, für Testszenarien zur Ausführung gebracht werden. Die Verwendung des Entwicklungswebservers ist jedoch mit Vorsicht zu genießen, da sich sein Verhalten im Detail von jenem, das von IIS an den Tag gelegt wird, unterscheidet.

2.4 Erste Schritte mit WCF

Nachdem in den vorangegangenen Abschnitten die Grundlagen zu Services und Serviceorientierung sowie zu WCF erläutert wurden, beschreibt dieser Abschnitt die Erstellung eines simplen WCF-Service. Dieser wird zunächst mit dem *BasicHttpBinding* und dann mit dem *WSHttpBinding* konfiguriert und im Entwicklungswebserver von Visual Studio zur Ausführung gebracht. Anschließend wird auf das *NetTcpBinding* umgestellt und ein benutzerdefinierter Host bereitgestellt. Zusätzlich wird für alle drei Entwicklungsstufen ein Client bereitgestellt, und die verwendeten SOAP-Nachrichten werden protokolliert.

2.4.1 Erstellen eines Web-Service-Projektes

Zur Erstellung eines auf HTTP basierenden Web-Service, der von einem Web- bzw. Applikationsserver gehostet werden soll, steht in Visual Studio die Projektvorlage *WCF Service Application* (Bild 2.1) zur Verfügung. Im hier beschriebenen Beispiel wird der Name *FlugService* vergeben.

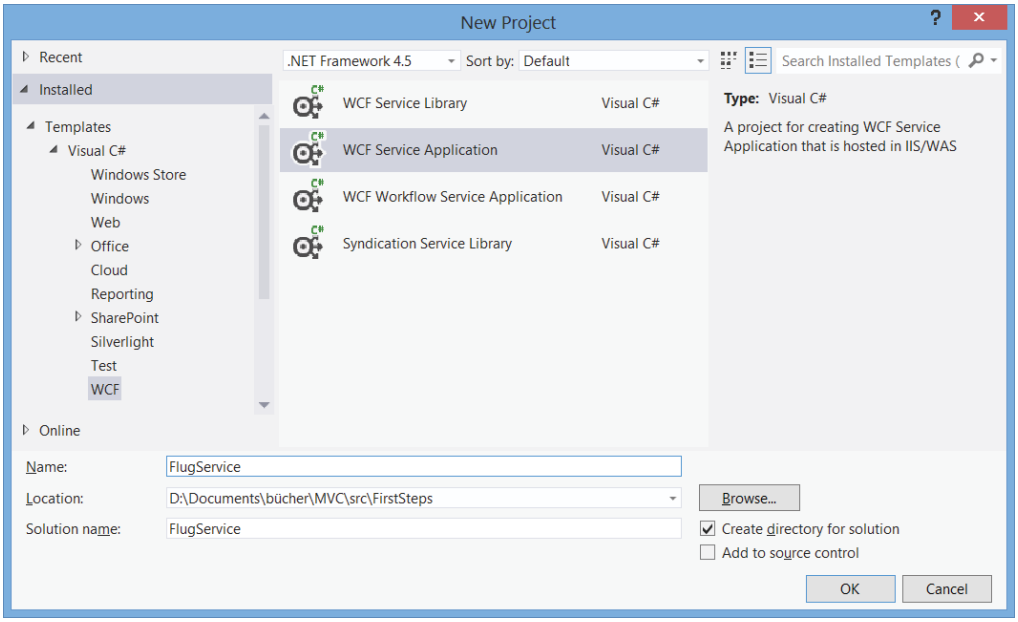

Bild 2.1 Erstellen eines neuen WCF-Projektes

2.4.1.1 Service hinzufügen

Wird mit der Projektvorlage *WCF Service Application* ein neues Projekt erzeugt, werden auch drei Dateien, die einen einfachen Service mit dem Namen *Service1* repräsentieren, eingerichtet. Diese tragen die Namen *IService1.cs*, *Service1.svc* und *Service1.svc.cs*, wobei Letztere im Solution-Explorer untergeordnet dargestellt wird. Um Probleme, die im Zuge einer Umbenennung entstehen können, zu vermeiden, zieht es der Autor vor, diese Dateien zu löschen und mit der Vorlage *WCF Service* (Bild 2.2) einen neuen Service, der den gewünschten Namen trägt, zu erzeugen. Im betrachteten Beispiel kommt der Name *FlugService* zum Einsatz. Dies führt dazu, dass drei neue Dateien angelegt werden: *IFlugService.cs*, *FlugService.svc* und *FlugService.svc.cs*. Erstere beinhaltet den Servicevertrag in Form eines Interface mit den bereitzustellenden Methoden; letztere die Serviceimplementierung. Die Datei *FlugService.svc* wird verwendet, um eine Verbindung zwischen einer über den Web Server bereitgestellten Datei und dem Service herzustellen, und beinhaltet lediglich einen Verweis auf die Service-Implementierung.

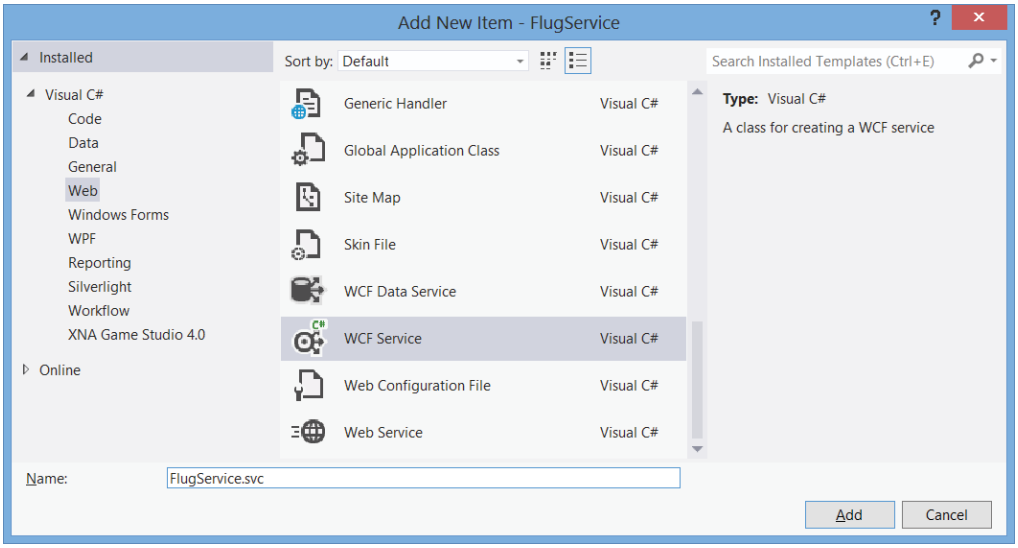

Bild 2.2 Erstellen eines neuen WCF-Services

2.4.1.2 Datenvertrag erstellen

Als Nächstes bietet sich die Erstellung eines Datenvertrages an. Datenverträge repräsentieren Klassen von Objekten, die zwischen Konsument und Services im Zuge der Kommunikation ausgetauscht werden. Dazu serialisiert WCF diese Objekte auf der Seite des Senders und sendet diese über das Netzwerk zum Empfänger, um sie dort wieder zu deserialisieren.

Um einen Datenvertrag zu erzeugen, wird eine neue Klasse *Flight* angelegt. Diese ist mit den zu übertragenden Eigenschaften auszustatten. Um die Klasse als Datenvertrag zu kennzeichnen, annotiert man sie mit dem Attribut *DataContract*. Die zu übertragenden Eigenschaften sind zusätzlich mit dem Attribut *DataMember* zu kennzeichnen (siehe Listing 2.1).

Listing 2.1 Datenvertrag für Flüge

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.ServiceModel;
using System.Runtime.Serialization;
namespace FlugService
{
    [DataContract(Namespace="www.softwarearchitekt.at/FlugService")]
    public class Flug
    {
        [DataMember]
        public String FlugNummer { set; get; }

        [DataMember]
        public String Von { set; get; }

        [DataMember]
        public String Nach { set; get; }
        [DataMember]
        public DateTime Abflug { set; get; }
        [DataMember]
        public double Preis { get; set; }
        public Flug(
                  String flugnummer,
                  String von,
                  String nach,
                  DateTime abflug,
                  double preis)
        {
            this.FulgNummer = flugnummer;
            this.Von = von;
            this.Nach = nach;
            this.Abflug = abflug;
            this.Preis = preis;
        }
    }
}
```

2.4.1.3 Service-Vertrag bereitstellen

Nachdem der benötigte Datenvertrag eingerichtet wurde, kann nun der Service-Vertrag mit den bereitzustellenden Service-Operationen implementiert werden. Dazu wird das Interface *IFlugService*, wie in Listing 2.2 gezeigt, erweitert. Um das Interface als Service-Vertrag zu kennzeichnen, annotiert man es mit dem Attribut *ServiceContract*. Die einzelnen Operationen, die vom Service angeboten werden, sind zusätzlich mit *ServiceOperation* zu annotieren. Analog zum Datenvertrag können Details der Serialisierung mit diesen Attributen gesteuert werden.

Listing 2.2 Service-Vertrag für den Flug-Service

```
[ServiceContract(Namespace="www.softwarearchitekt.at/FlugService")]
public interface IFlugService
{
     [OperationContract]
      List<Flug> FindFlights(
            String von,
            String nach,
            DateTime datum);
}
```

Als Nächstes wird die Service-Implementierung mit Leben erfüllt. Dazu ist lediglich sicherzustellen, dass die Klasse in der Datei *FlugService.svc.cs* das Service-Interface implementiert. Listing 2.3 zeigt eine beispielhafte Implementierung, welche die abzufragenden Flüge in einer statischen Liste vorhält.

Listing 2.3 Service-Implementierung für den Flug-Service

```
 public class FlugService : IFlugService
    {
        private static List<Flug> fluege = null;
        public FlugService()
        {
            if (fluege == null)
            {
                fluege = new List<Flug>();
                fluege.Add(new Flug("LH0815", "Graz", "Frankfurt",
new DateTime(2009, 12, 08, 07, 00, 00), 200));
                fluege.Add(new Flug("LH0815", "Graz", "Frankfurt",
new DateTime(2009, 12, 08, 14, 00, 00), 300));
                fluege.Add(new Flug("LH0815", "Graz", "Frankfurt",
new DateTime(2009, 12, 08, 15, 00, 00), 400));
                fluege.Add(new Flug("LH4711", "Graz", "Mallorca",
new DateTime(2009, 12, 08, 15, 00, 00), 90));
            }
        }
        public List<Flug> FindFlights(String von, String nach, DateTime
datum)
        {
            List<Flug> result = new List<Flug>();
            foreach (Flug f in fluege)
            {
                if (f.Von == von
                        && f.Nach == nach
                        && f.Abflug.Date == datum)
                {
                    result.Add(f);
                }
            }
            return result;
        }
    }
```

2.4.1.4 Überlegungen zur Standardkonfiguration

In früheren Versionen von WCF hätte nun der implementierte Service in der *web.config* konfiguriert werden müssen. Im Zuge dessen wäre ein Service-Endpunkt, der sich auf ein Binding und einen Service-Vertrag abstützt, einzurichten gewesen. Ab Version 4 ist dies nicht mehr nötig, da WCF nun für alle Services, die nicht explizit konfiguriert wurden, einen Standardendpunkt einrichtet. Dazu wird aus der zu verwendenden Adresse ein Standard-Binding abgeleitet und mit diesem ein Standardendpunkt mit dieser Adresse erzeugt. Da das hier beschriebene Projekt in einem Webserver gehostet wird, ist die Adresse des Service über die SVC-Datei vorbestimmt. Da diese mit *http://* beginnt, wird standardmäßig das *BasicHttpBinding* herangezogen.

Somit beschränkt sich die Datei *web.config* auf wenige vordefinierte Zeilen (siehe Listing 2.4). Die für WCF relevanten Einträge befinden sich unter *system.serviceModel*. Hier findet sich ein Service-Behavior wieder. Da dieser keinen Namen aufweist, kommt er für alle Services, die auf keinen anderen Service-Behavior explizit verweisen, zum Einsatz. Das gilt auch für den in diesem Abschnitt beschriebenen Flug-Service. Das Element *serviceMetadata* steuert die Veröffentlichung von Metadaten über den Service. Da die Eigenschaft *httpGetEnabled* den Wert *true* aufweist, veröffentlicht WCF Metadaten in Form eines WSDL-Dokuments über HTTP. Um diese Datei abzurufen, ist lediglich die Endung *?wsdl* an die Service-URL anzuhängen, sofern mit den Eigenschaften von *serviceDebug* nichts anderes festgelegt wurde.

Die Eigenschaft *includeExceptionDetailInFaults* im Element *serviceDebug* legt fest, ob Details zu Ausnahmen als Teil der von WCF generierten SOAP-Faults übertragen werden sollen. Im Zuge der Entwicklung bietet es sich an, diese Option wie in Listing 2.4 auf *true* zu setzen. Bei Produktivsystemen sollte diese Option jedoch durch Angabe von *false* deaktiviert werden, da die Details von eventuell aufgetretenen Ausnahmen wertvolle Informationen für potenzielle Angreifer darstellen können.

Listing 2.4 WCF-Konfiguration in der Datei web.config

```
[…]
<system.serviceModel>
<behaviors>
      <serviceBehaviors>
       <behavior>
            <serviceMetadata httpGetEnabled="true"/>
            <serviceDebug includeExceptionDetailInFaults="true"/>
       </behavior>
      </serviceBehaviors>
</behaviors>
<serviceHostingEnvironment multipleSiteBindingsEnabled="true" />
</system.serviceModel>
[…]
```

2.4.1.5 Service starten

Da Fehler in der Konfiguration erst beim Start des Service erkannt werden, bietet es sich an, nach jeder Änderung den Service zur Ausführung zu bringen, bevor Servicekonsumenten entwickelt bzw. angepasst werden. Im Zuge dessen sollte die Service-Implementierung nicht im Solution-Explorer markiert sein, da in diesem Fall bereits ein generischer Test-Client von Visual Studio gestartet wird.

Im Zuge der Projektausführung, die mit *Debug* | *Start Debugging* angestoßen werden kann, startet Visual Studio den Entwicklungswebserver. Dieser mach sich als Symbol links von der Uhr in der Taskleiste bemerkbar. Zusätzlich wird ein Browserfenster geöffnet. Nachdem die SVC-Datei ausgewählt wurde, erscheint, sofern die Konfiguration keinen Fehler aufweist, eine Begrüßungsseite (Bild 2.3). Ansonsten wird man mit einer Ausnahme konfrontiert.

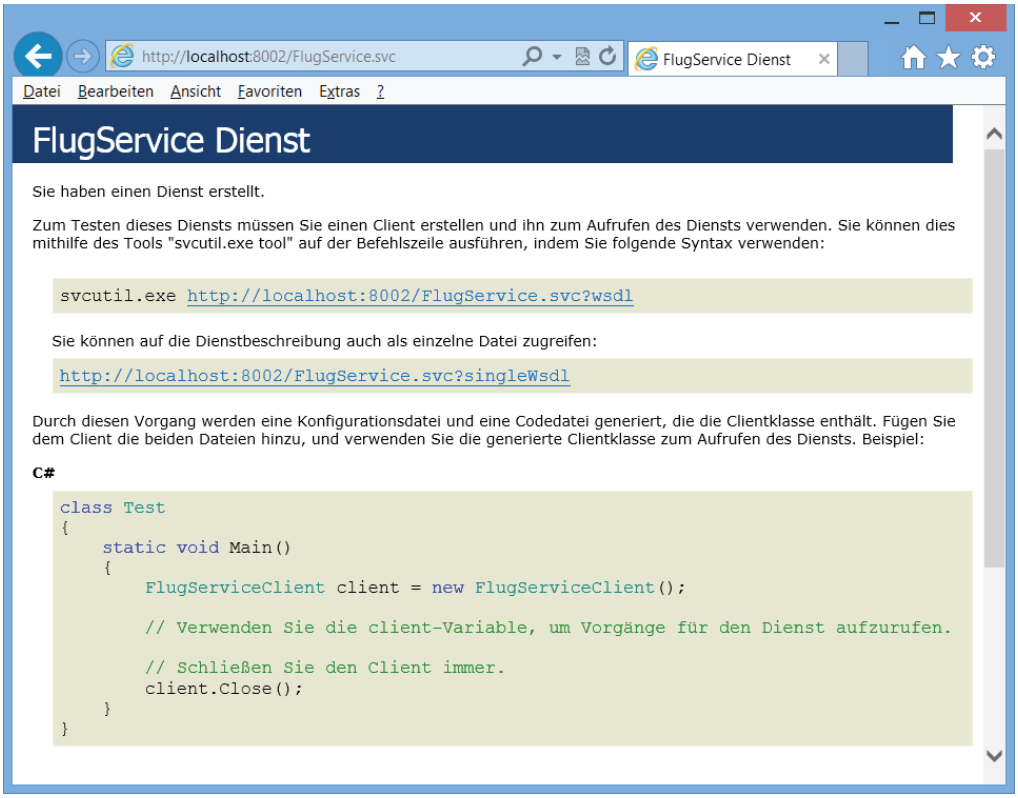

Bild 2.3 Begrüßungsseite

Ein Klick auf einen der beiden Links in der Begrüßungsseite führt zum WSDL-Dokument, das WCF für den Service generiert hat und zur Erstellung von Proxys am Client herangezogen werden kann (Bild 2.4). Der Unterschied zwischen diesen beiden Links besteht darin, dass der erste Link, welcher seit den ersten Tagen von WCF zur Verfügung steht, zu einem WSDL-Dokument führt, welches wiederum auf weitere von WCF generierten WSDL- und XML-Schema-Dokumente verweist. Diese Aufteilung reduziert die Komplexität der einzelnen Dokumente, macht es für den Entwickler jedoch schwieriger, die gesamten Informationen herunterzuladen. Aus diesem Grund führt der zweite Link, welcher seit .NET 4.5 angeboten wird, zu einer Version dieser WSDL-Datei, welche sämtliche Informationen beinhaltet und somit nicht auf andere Dokumente verweisen muss.

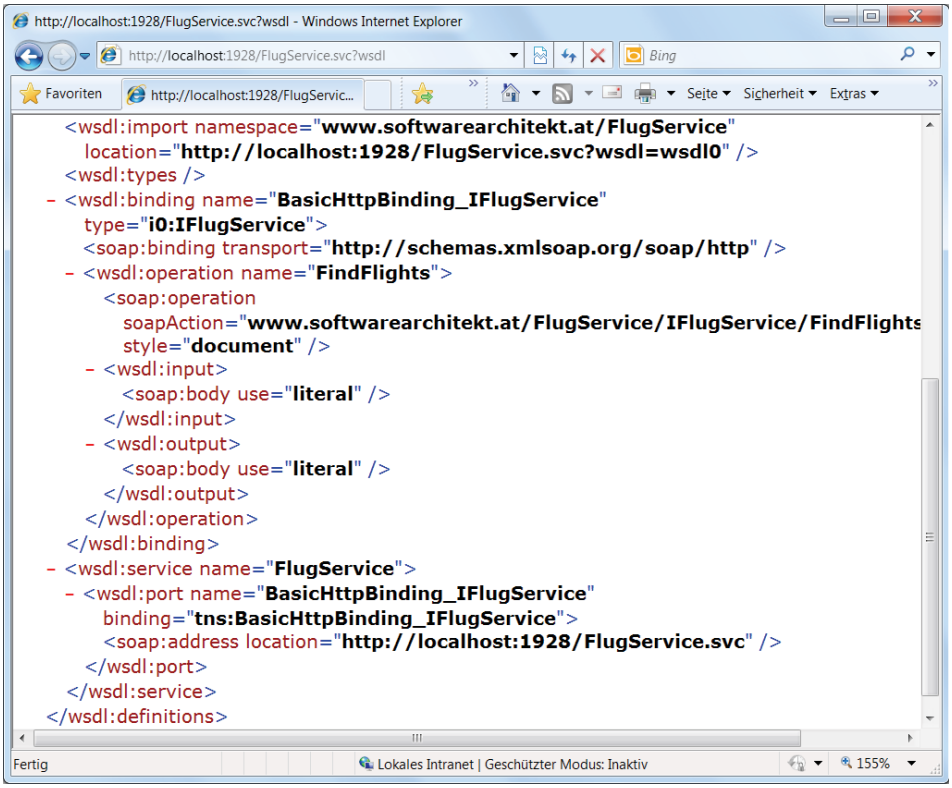

Bild 2.4 Generiertes WSDL-Dokument

2.4.1.6 Service mit generischem Client testen

Um den erstellten Service mit dem im Lieferumfang von WCF enthaltenen generischen Test-Client zu testen, wird der Service im Solution Explorer markiert und das Projekt zur Ausführung gebracht. Nun sollte der Test-Client gestartet werden. Ist dem nicht so, muss der *Visual Studio Command Prompt* aufgerufen werden (*Start* | *Alle Programme* | *Visual Studio* | *Visual Studio Tools* | *Visual Studio Command Prompt*). Anschließend kann der Test-Client durch Eingabe von *wcfTestClient* zur Ausführung gebracht werden. Nach dem Start über den Command Prompt gibt man die URL des zu testenden Service manuell an (*File* | *Add Service*). Beim Start aus Visual Studio heraus ist dies nicht notwendig.

Nun wird links die implementierte Service-Operation ausgewählt. Anschließend gibt man rechts die zu verwendenden Parameterwerte an und klickt auf *Invoke*. Daraufhin wird die Service-Operation angestoßen, und das Ergebnis wird rechts unten angezeigt (Bild 2.5). Per Klick auf *XML* im unteren Bereich können auch die übertragenen SOAP-Nachrichten eingesehen werden (Bild 2.6).

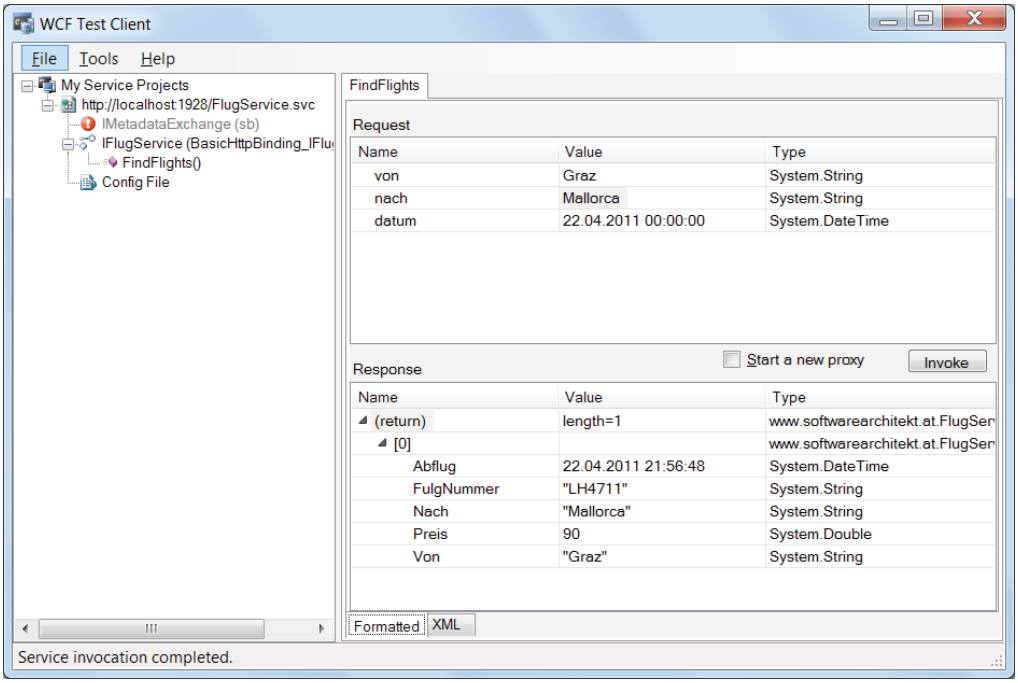

Bild 2.5 Aufruf der implementierten Service-Operation über den Test-Client

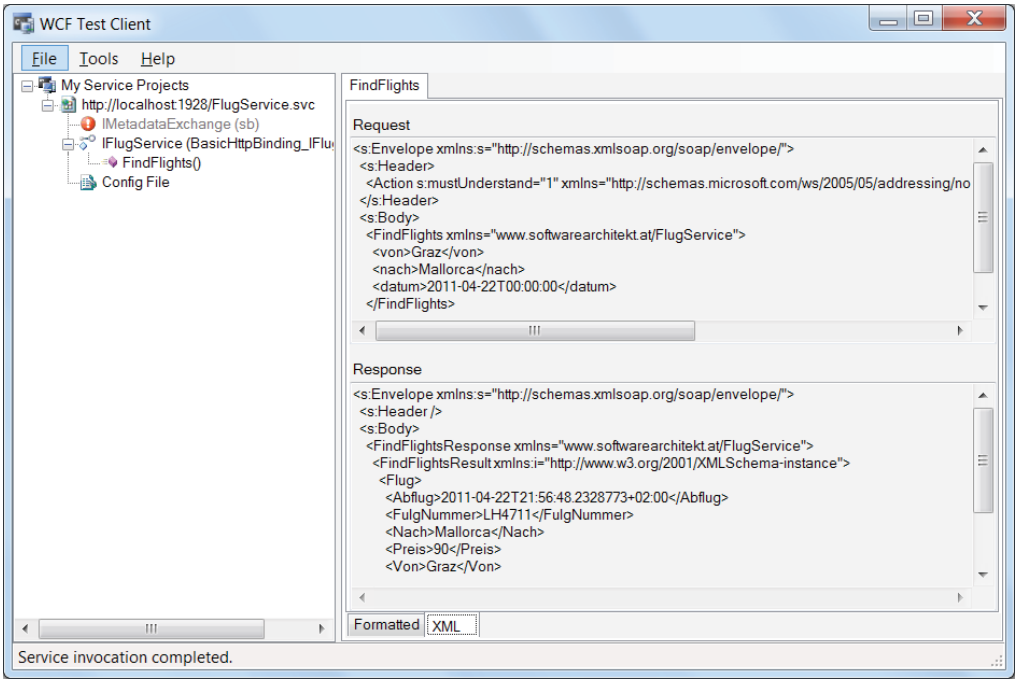

Bild 2.6 Ansicht der übertragenen SOAP-Nachrichten im Test-Client

2.4.2 Web-Service mit Client konsumieren

Um einen einfachen Client, der den Web-Service konsumiert, bereitzustellen, wird der Solution ein neues Projekt vom Typ *Windows Forms Application* hinzugefügt (*File* | *New* | *Project*). Als Name wird *SimpleClient* vergeben.

HINWEIS: Wenn gerade ein Projekt über Visual Studio ausgeführt wird, muss dieses beendet werden (*Debug* | *Stop Debugging*), bevor ein weiteres Projekt innerhalb der Solution angelegt werden kann.

In diesem Projekt gibt man an, eine neue Service-Referenz hinzufügen zu wollen (Rechtsklick auf *Service References im Solution Explorer* | *Add Service Reference*). Anschließend wird die Adresse der WSDL-Datei des gewünschten Service unter *Address* eingetragen. Alternativ dazu kann der Dialog per Klick auf *Discover* veranlasst werden, diese Adresse selbst anhand der in der Solution vorliegenden Informationen herauszufinden (Bild 2.7).

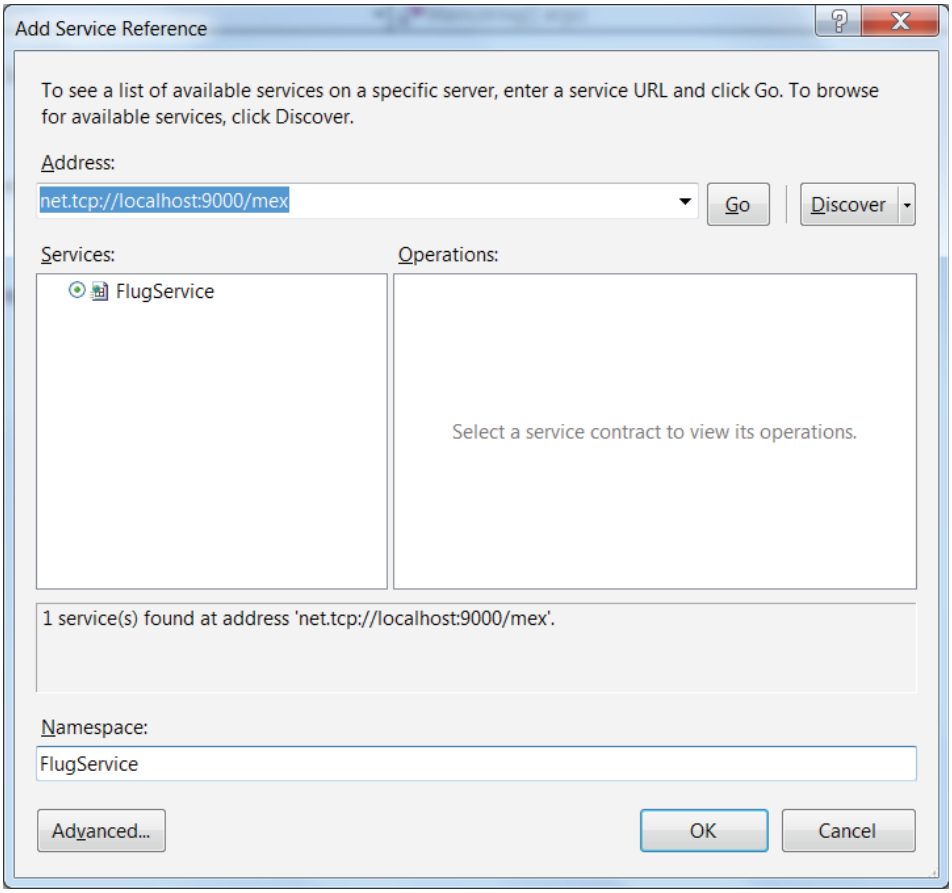

Bild 2.7 Hinzufügen einer Service-Referenz

Ein Klick auf die Schaltfläche *Advanced* führt zu einem Dialog, der erweiterte Einstellungen zulässt. Hier wird unter *Collection type* der Eintrag *System.Collections.Generic.List* ausgewählt (Bild 2.8). Dies hat zur Folge, dass der generierte Proxy für alle in der WSDL-Datei beschriebenen Auflistungen generische Listen spendiert bekommt. Standardmäßig werden für Auflistungen Arrays verwendet, was sich aufgrund ihres statischen Charakters jedoch als unhandlich herausgestellt hat.

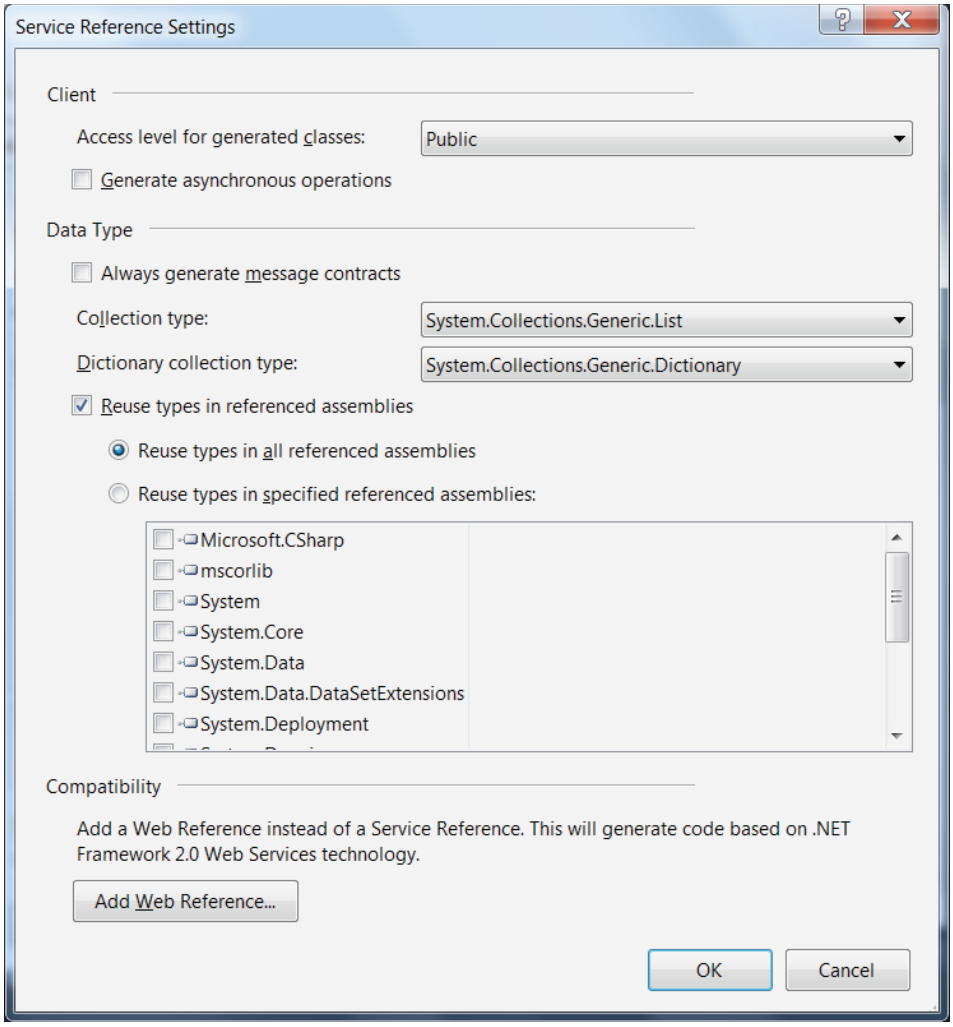

Bild 2.8 Festlegen erweiterter Einstellungen beim Hinzufügen einer Service-Referenz

Darüber hinaus kann der Entwickler durch Aktivieren der Option *Reuse types in referenced assemblies* angeben, dass er am Client zur Verfügung stehende Service- und Daten-Verträge, welche sich in eingebundenen Assemblies befinden, verwenden möchte, sofern diese existieren. Aktiviert er diese Option nicht oder sind diese Konstrukte nicht am Client verfügbar, generiert Visual Studio diese anhand der Beschreibungen im WSDL-Dokument. Diese

Option erlaubt den Einsatz sogenannter *Shared Contracts*. Das sind Verträge, die der Entwickler in eine eigene Assembly, welche sowohl in das Service- als auch in das Client-Projekt eingebunden wird, auslagert. Somit können zum Beispiel Datenverträge mit Methoden, die Werte aus anderen Werten berechnen, versehen werden. Da solche Methoden nicht durch WSDL beschrieben werden können, stünden diese Methoden am Client nicht zur Verfügung, wenn der Entwickler diese Datenverträge aus dem WSDL-Dokument generieren würde.

Nachdem die beiden Dialoge bestätigt wurden, wird eine Service-Referenz hinzufügt, Konfigurationseinträge werden hinterlegt sowie ein Proxy erstellt.

Ein Blick in die Konfigurationsdatei *app.config* lässt unter *system.serviceModel* einige Einträge zur Steuerung des soeben generierten Proxys erkennen, darunter eine Binding-Konfiguration, welche die Standardwerte des verwendeten *BasicHttpBindings* widerspiegelt (Listing 2.5). Aus Gründen der Übersichtlichkeit und da Details der Binding-Konfiguration in 3 besprochen werden, ist dieser Bereich im betrachteten Listing lediglich angedeutet. Zusätzlich findet sich im Element *Client* ein Client-Endpunkt, der auf den bereitgestellten Service verweist, wieder. Dieser wurde mit der jeweiligen Adresse, dem Binding *BasicHttpBinding* und dem aus dem WSDL-Dokument generierten Service-Vertrag (Contract) konfiguriert (ABC). Zusätzlich verweist dieser Eintrag auf die zuvor erwähnte Binding-Konfiguration, und als Name kommt *BasicHttpBinding_IFlugService* zum Einsatz.

Listing 2.5 Generierte Client-Konfiguration

```
[…]
<system.serviceModel>

    <bindings>
        <basicHttpBinding>
            <binding name="BasicHttpBinding_IFlugService">
                […]
            </binding>
        </basicHttpBinding>
    </bindings>
    <client>
        <endpoint
            address="http://localhost:1928/FlugService.svc"
            binding="basicHttpBinding"
            contract="FlugService.IFlugService"
            bindingConfiguration="BasicHttpBinding_IFlugService"
            name="BasicHttpBinding_IFlugService" />
    </client>
</system.serviceModel>
[…]
```

HINWEIS: Falls sich der verwendete Rechner normalerweise in einer Domäne befindet, dies jedoch während der Ausführung des Services nicht der Fall ist, sollte ein eventuelles *identity*-Element, welches sich ggf. innerhalb von *endpoint* befindet, samt allen untergeordneten Elementen, zu Testzwecken aus der Konfiguration entfernt werden. Weitere Informationen hierzu finden Sie in Kapitel 4, „Sicherheit von WCF-Diensten“.

Um über den generierten Proxy nun den bereitgestellten Service aufzurufen, fügt man dem Formular Form1 eine *DataGridView* sowie eine *Schaltfläche* hinzu (Bild 2.9).

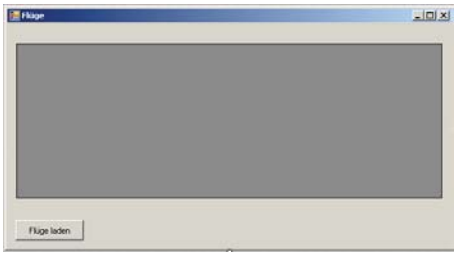

Bild 2.9 Demo-Client in der Entwurfsansicht

Anschließend fügt man der Schaltfläche eine Ereignisbehandlungsroutine für das Click-Ereignis hinzu (Listing 2.6). Diese instanziiert den Proxy, der den Namen *FlugServiceClient* trägt. Danach ruft der Proxy mit der Methode *FindFlights* die gleichnamige Service-Operation auf. Die zurückgelieferten Daten werden an die *DataGridView* gebunden und der Proxy mittels *Close* geschlossen. Im Falle einer Exception, die den Proxy in den Zustand *Faulted* versetzt, ist hingegen per Definition *Abort* aufzurufen, um den Proxy zu schließen.

Listing 2.6 Aufruf des generierten Proxys

```
using SimpleClient.FlugService;
[…]
private void button1_Click(object sender, EventArgs e)
{
    FlugServiceClient c = new FlugServiceClient();
    try
    {
        List<Flug> result;
        result = c.FindFlights("Graz", "Frankfurt", new DateTime(2009, 12,
8));
        this.dataGridView1.DataSource = result;
        c.Close();
    }
    catch
    {
        c.Abort();
    }
}
```

HINWEIS: Weist ein Service mehrere Endpunkte auf, muss der Name des gewünschten Client-Endpunktes, der in der Konfiguration vergeben wurde, als Argument an den Konstruktor des Proxys übergeben werden.

Nun können der Client und der Service gemeinsam zur Ausführung gebracht werden. Dazu ist in der Konfiguration der Solution (Rechtsklick auf *Solution im Solution Explorer* | *Properties*) unter *Startup Project* die Option *Multiple startup projects* zu wählen und für jedes der beiden Projekte die Aktion *Start* anzugeben. Beim Start der Solution (*Debug* | *Start Debug-*

ging) werden nun beide Projekte zur Ausführung gebracht. Ein Klick auf die Schaltfläche im Demo-Client sollte zur Anzeige einiger Flüge führen.

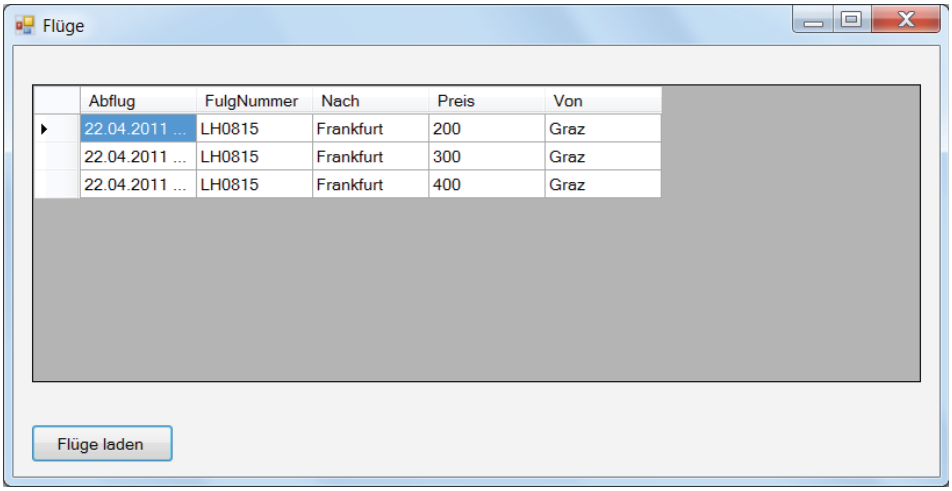

Bild 2.10 Abrufen von Flügen über den Demo-Client

HINWEIS: .NET liegt ein Kommandozeilentool *svcutil.exe*, das zum Beispiel über den Visual Studio-Command Prompt ausgeführt werden kann, bei. Damit lassen sich auch, unter Angabe zahlreicher Optionen, Proxys aus WSDL-Dateien generieren.

Alternativ dazu können die damit generierten Service- und Daten-Verträge auch für die Erstellung eines eigenen Services, der sich an der vorliegenden WSDL-Datei orientiert, verwendet werden.

2.4.3 Mit Laufzeit-Proxy auf Service zugreifen

In Abschnitt 2.4.2 wurde gezeigt, wie ein Proxy zum Zugriff auf einen Service in Visual Studio generiert werden kann. Alternativ dazu kann ein solcher Proxy auch zur Laufzeit erstellt werden. In diesem Fall müssen zunächst die Service-, Nachrichten- und Datenverträge am Client nachgebildet oder diesem über eine gemeinsame Assembly bereitgestellt werden. Zusätzlich ist man bei dieser Vorgehensweise angehalten, die benötigten Konfigurationseinträge manuell zu erstellen. Dem Client-Endpunkt muss dabei über das Attribut *Name* ein Name zugewiesen werden.

In der Applikation ist eine *ChannelFactory*, die mit dem gewünschten Service-Vertrag zu parametrisieren ist, einzurichten (Listing 2.7). Dabei ist an den Konstruktor der in der Konfiguration festgelegte Endpoint-Name zu übergeben. Alternativ dazu kann über die Eigenschaft *Endpoint* dieser auch programmatisch konfiguriert werden. Um den Laufzeitproxy zu erzeugen, wird anschließend die Methode *CreateChannel* aufgerufen.

Listing 2.7 Einsatz eines Laufzeit-Proxys

```
ChannelFactory<IFlugService> cf;
IFlugService proxy;
cf = new ChannelFactory<IFlugService>("BasicHttpBinding_IFlugService");
proxy = cf.CreateChannel();
var result = proxy.FindFlights("Graz", "Frankfurt", new DateTime(2009, 12,
8));
proxy.
this.dataGridView1.DataSource = result;
((IDisposable)proxy).Dispose();
cf.Close();
```

2.4.4 Service zur Verwendung von ws2007HttpBinding konfigurieren

Der in den letzten Abschnitten beschriebene Service wurde nicht explizit konfiguriert. Deswegen hat WCF eine Standardkonfiguration, die bei HTTP-basierten Services die Verwendung des *BasicHttpBinding* vorsieht, herangezogen. Dieser Abschnitt zeigt anhand einer beispielhaften Konfiguration, wie dieser Service für die Verwendung des *ws2007HttpBinding* konfiguriert werden kann. Dazu wird die Konfiguration aus Listing 2.8 in der Datei *app.config* eingetragen. Mit dem Attribut *name* im Element *service* wird auf den zu konfigurierenden Service verwiesen, indem hier der vollständige Name der Service-Implementierung (*Namensraum.Klassenname*) eingetragen wird. Das Attribut *behaviorConfiguration* beinhaltet den Namen der weiter unten hinterlegten Behavior-Konfiguration und verweist somit auf diese. Würde der Service auf keine Behavior-Konfiguration explizit verweisen, käme die Standard-Behavior-Konfiguration zum Einsatz. Dabei handelt es sich um eine Behavior-Konfiguration, für die kein Name vergeben wurde. Das Element *Endpoint* definiert einen Endpunkt für den Service. Da die Adresse im betrachteten Beispiel durch die SVC-Datei vorbestimmt ist, wird das Attribut *Address* auf einen Leerstring gesetzt. Ein hier eingetragener Wert würde zur Bildung der Endpunkt-Adresse an die Adresse der SVC-Datei anhängt werden. Als Binding kommt *ws2007HttpBinding* zum Einsatz; als Service-Vertrag das Interface *FlugService.IFlugService*. Um Letzteres festzulegen, wird für das Attribut *Contract* der vollständige Name (*Namensraum.Klassenname*) des erwähnten Interface eingetragen. Zusätzlich verweist der Endpunkt über das Attribut *bindingName* auf die weiter unten definierte Binding-Konfiguration mit dem Namen *MyBindingConfiguration*. Auch hier gilt, dass standardmäßig auf eine eventuell vorhandene Binding-Konfiguration ohne Namen verwiesen wird.

Um das Beispiel einfach zu halten, deaktiviert die Binding-Konfiguration die standardmäßig bei *ws2007HttpBinding* aktivierten Sicherheitseinstellungen, zumal sich Kapitel 4, „Sicherheit von WCF-Diensten“, um diesen Aspekt kümmert. Die Behavior-Konfiguration beinhaltet die bereits aus Abschnitt 2.4.1.4 bekannten Einstellungen.

Listing 2.8 Konfiguration des Flug-Service zur Verwendung des ws2007HttpBinding

```
[…]
<system.serviceModel>
      <services>
            <service
```

```
                behaviorConfiguration="MyServiceBehavior"
                name="FlugService.FlugService">

                 <endpoint
                     address=""
                     binding="ws2007HttpBinding"
                     contract="FlugService.IFlugService"
                     bindingName="MyBindingConfiguration"/>
            </service>

      </services>

      <bindings>
            <ws2007HttpBinding>
                  <binding name="MyBindingConfiguration">
                        <security mode="None"></security>
                  </binding>
            </ws2007HttpBinding>
      </bindings>

      <behaviors>
            <serviceBehaviors>
                  <behavior name="MyServiceBehavior">
                        <serviceMetadata httpGetEnabled="true"/>
                        <serviceDebug includeExceptionDetailInFaults="true"/>
                  </behavior>
            </serviceBehaviors>
      </behaviors>

</system.serviceModel>
[…]
```

Um auf eventuelle Konfigurationsfehler hingewiesen zu werden, sollte der Service nun gestartet und die SVC-Datei über den Browser aufgerufen werden. Falls dies korrekt war, erscheint die bereits besprochene Willkommensseite.

Danach muss noch die Service-Referenz im Demo-Client aktualisiert werden (Rechtsklick auf den Namen der *Service-Referenz* im *Solution-Explorer* | *Update Service Reference*). Dies aktualisiert die Client-Konfiguration. Anschließend kann die Solution wie unter 2.4.1.5 beschrieben zur Ausführung gebracht und über den Client können Flüge abgerufen werden.

2.4.5 NetTcpBinding und Self-Hosting

Dieser letzte Abschnitt zeigt, wie der in den vorhergehenden Abschnitten entwickelte Service zusammen mit dem *netTcpBinding* eingesetzt werden kann. Da der Entwicklungswebserver lediglich auf HTTP basierende Bindings unterstützt, wird hierzu ein eigener Service-Host erstellt.

2.4.5.1 Service erstellen

Für die Demonstration des hier vorgestellten Szenarios wird zunächst eine neue Solution mit der Projektvorlage *WCF Service Library* erstellt. Als Name wieder abermals *FlugService* vergeben (Bild 2.11).

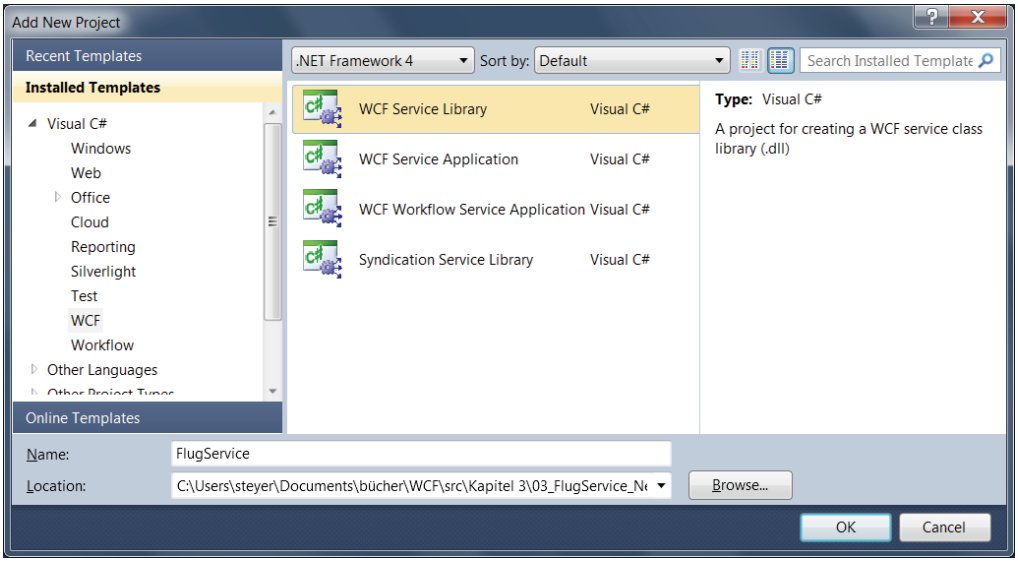

Bild 2.11 Erzeugen einer Solution mit Projekt vom Typ WCF Service Library

Um den Datenvertrag, den Service-Vertrag und die Service-Implementierung nicht erneut implementieren zu müssen, bietet es sich an, die Dateien *Flug.cs*, *IFlugService.cs* sowie *IFlugService.svc.cs* aus dem in den vorhergehenden Abschnitten beschriebenen Beispiel zu kopieren. Alternativ dazu können die Inhalte dieser Dateien auch aus Listing 2.1, Listing 2.2 und Listing 2.3 entnommen werden.

2.4.5.2 Service-Host erstellen

Zur Implementierung des benutzerdefinierten Service-Hosts wird dem Projekt nun eine Kommandozeilen-Applikation mit dem Namen *Host* hinzugefügt und ein Verweis auf die Assembly *System.ServiceModel*, die Klassen der WCF bereitstellt, erzeugt (Rechtsklick auf *References* im *Solution Explorer* | *Add Reference* | *Auswahl von System.ServiceModel im Registerblatt .NET*). Zusätzlich wird auf die zuvor erstellte WCF Service Library verwiesen (Rechtsklick auf *References* im *Solution Explorer* | *Add Reference* | *Auswahl des Projektes FlugService im Registerblatt Projects*). Danach wird der Service-Host in der ausführbaren Klasse *Program* implementiert (Listing 2.9). Diese Klasse instanziiert die Klasse *ServiceHost* und übergibt den Typ der Service-Implementierung des bereitzustellenden Service an ihren Konstruktor. Anschließend ruft sie die Methode *Open* auf und startet somit den Service.

Listing 2.9 Implementierung des Service-Hosts

```
[…]
class Program
{
    static void Main(string[] args)
    {
        using(ServiceHost host =
                new ServiceHost(typeof(FlugService.FlugService))) {
            host.Open();
```

```
            Console.WriteLine("Service gestartet.");
            Console.WriteLine("[Enter] stoppt den Service.");
            Console.ReadLine();
        }
    }
}
[…]
```

Damit der selbst implementierte Service-Host seiner Aufgabe nachkommen kann, benötigt er noch eine Konfiguration für den Service. Dazu ist eine Applikationskonfigurationsdatei unter Verwendung des gleichnamigen Item-Templates anzulegen. Als Name wird *app.config* vergeben. Die vorzunehmenden Konfigurationseinträge können aus Listing 2.10 entnommen werden.

Da bei Verwendung eines selbst implementierten Hosts im Gegensatz zum Hosting in einem Webserver die Basisadresse nicht vorbestimmt ist, gibt das Element *baseAddresses* eine solche an. Als URL-Schema wird, wie bei auf TCP basierenden WCF-Services üblich, *net.tcp* verwendet. Der erste veröffentlichte Endpunkt stützt sich auf das *NetTcpBinding* sowie auf den bereitgestellten Service-Vertrag ab. Die Adresse ist leer, weswegen dieser Endpunkt direkt über die Basisadresse erreicht werden kann.

Da keine HTTP-basierte Basisadresse zur Verfügung steht, wurde die Option *httpGetEnabled* im Standard-Service-Behavior deaktiviert. Stattdessen weist der Service einen weiteren Endpunkt auf, der das Abrufen von Metadaten via TCP durch Anwendung des Standards *WS-MetadataExchange* erlaubt. Als Binding kommt *mexTcpBinding* zum Einsatz; als Adresse *mex*. Somit kann dieser Endpunkt erreicht werden, indem an die Basisadresse */mex* angehängt wird. Das Attribut *contract* verweist auf das in WCF inkludierte Interface *IMetadataExchange*. Damit dieses durch den Host gefunden und auch implementiert wird, muss sich das Element *serviceMetadata* im Service-Behavior wiederfinden.

Listing 2.10 Konfiguration zur Verwendung von netTcpBinding und Self-Hosting

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
 <system.serviceModel>
    <services>
      <service name="FlugService.FlugService">
        <endpoint
          address=""
          binding="netTcpBinding"
          contract="FlugService.IFlugService" />
        <endpoint
          binding="mexTcpBinding"
          address="mex"
          contract="IMetadataExchange" />
        <host>
          <baseAddresses>
            <add baseAddress="net.tcp://localhost:9000"/>
          </baseAddresses>
        </host>
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
```

```
        <behavior>
          <serviceMetadata httpGetEnabled="false"/>
          <serviceDebug includeExceptionDetailInFaults="true"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

Um die Konfiguration auf Fehler zu prüfen, sollte der Host ausgeführt werden. Tritt dabei kein Fehler auf, wird eine Ausnahme ausgelöst. Geht alles gut, ergibt sich das aus Bild 2.12 ersichtliche Bild.

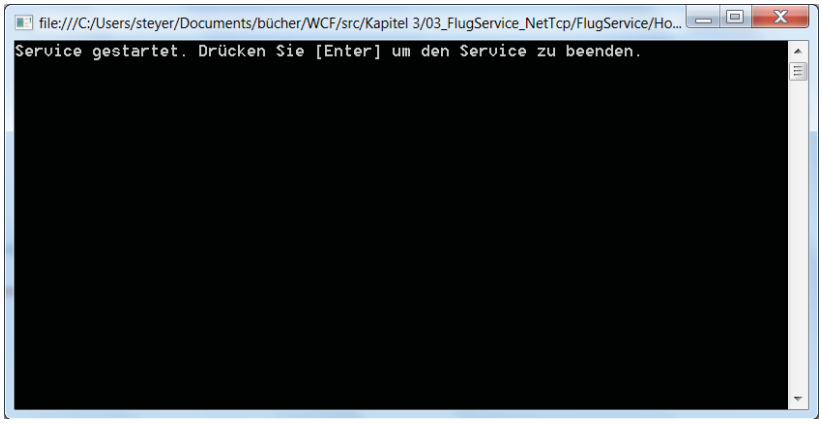

Bild 2.12 Selbst implementierter Host

HINWEIS: Ab Windows Vista müssen Benutzer vom Administrator eine Erlaubnis erhalten, um Services über HTTP bereitstellen zu dürfen. Deswegen würde der hier gezeigte Service-Host mitunter einen Fehler auslösen, wenn er sich anstatt auf TCP auf HTTP abstützte. Zum Erteilen dieser Rechte, steht das Dienstprogramm *netsh* zur Verfügung. Die folgende Anweisung würde zum Beispiel dem Benutzer *DOMAIN\user* das Recht geben, einen HTTP-basierenden Service über Port 8080 bereitzustellen: *netsh http add urlacl url=http://+:8080 user=DOMAIN\user*

2.4.5.3 Service konsumieren

Um den selbst gehosteten Service zu konsumieren, muss dieser außerhalb von Visual Studio über den Windows-Explorer oder über Visual Studio mit dem Befehl *Debug | Start without Debugging* gestartet werden. Der Grund dafür liegt in der Tatsache, dass es die Standardeinstellungen von Visual Studio nicht erlauben, eine Applikation zu erweitern, während sich eine andere im Debug-Modus befindet.

Anschließend kann einem Client-Projekt eine Service-Referenz, die auf den bereitgestellten Service verweist, hinzufügt werden (siehe Abschnitt 2.4.2). Die zu erfassende Adresse ver-

weist in diesem Fall jedoch nicht auf ein WSDL-Dokument, sondern auf den konfigurierten MEX-Endpunkt (Bild 2.13).

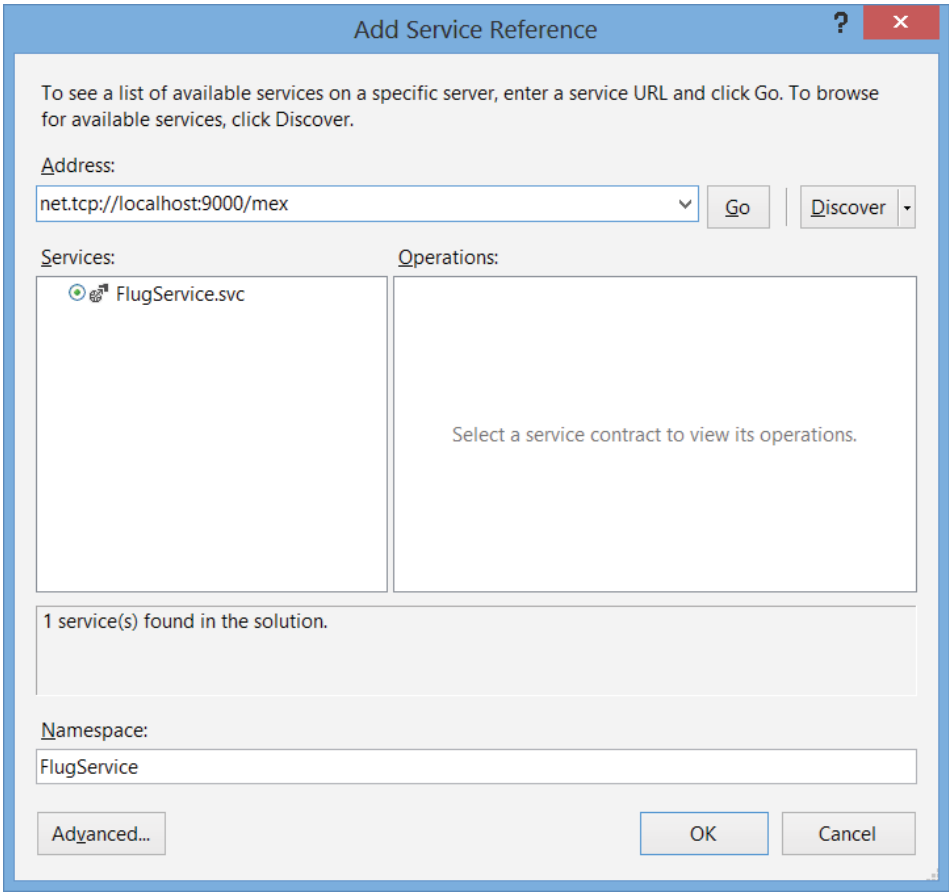

Bild 2.13 Service-Referenz unter Verwendung eines MEX-Endpunktes hinzufügen