Usługi sieciowe
laboratorium
2019

K.M. Ocetkiewicz, T. Goluch

1. Wstęp

Usługa sieciowa jest zbiorem funkcji udostępnianych przez serwer. Jest to kolejne podejście do problemu zdalnego wołania procedur, w tym wypadku kładące większy nacisk na komunikację poprzez sieć (zdalne wołanie procedur ma zastosowanie do wołań pomiędzy procesami na jednej maszynie, w przypadku usług sieciowych oczywiście jest to także wykonalne, ale z powodu narzutów komunikacyjnych niepraktyczne).

Z usługami sieciowymi wiąże się szereg pojęć, które pokrótce omówimy. Część z nich może dawać wrażenie obiektowego modelu korzystania z usług sieciowych. W rzeczywistości modelowi temu bliżej jest do zwykłego zdalnego wołania procedur niż do obiektowości, którą mieliśmy w COM.

Serwer

Serwer jest bytem udostępniającym zbiór funkcji. Funkcje te są wykonywane przez komputer, na którym serwer jest uruchomiony. W zależności od kontekstu, przez serwer będziemy określali zarówno komputer który udostępnia usługę, jak i proces czy nawet pojedynczy obiekt, które udostępniają czy implementują daną usługę.

Klient

Klient jest procesem korzystającym z usług serwera. Wywołuje on metody z usługi, lecz ich wykonanie odbywa się po stronie serwera. Klient korzysta z usługi poprzez proxy – obiekt w przestrzeni adresowej klienta udający usługę i udostępniający jej funkcje.

Usługa

Usługa jest zbiorem zgrupowanych razem funkcji, pełni zatem funkcję analogiczną do interfejsu. I rzeczywiście, od pewnego momentu zaczniemy usługę utożsamiać z interfejsem i używać tych pojęć zamiennie. Grupowanie funkcji w usługę ma znaczenie dla klienta, który będzie posiadał jedno proxy na usługę. Jest także istotne podczas określania czasu życia – definiujemy go dla usługi, a zatem dla wszystkich dostępnych w niej funkcji.

Czas życia usługi

Analogia pomiędzy usługą a interfejsem sięga dalej. Tak jak obiekty implementowały interfejsy, tak i tu będziemy musieli zaimplementować funkcje z usługi w pewnym obiekcie. Czas życia usługi określa w rzeczywistości czas życia obiektu implementującego usługę. Domyślnym czasem życia w większości przypadków jest „jeden obiekt na wołanie". Oznacza to, że każde wywołanie funkcji z usługi powoduje utworzenie obiektu serwera, wywołanie na nim odpowiedniej metody a następnie zniszczenie serwera. Inne typowe czasy życia to „jeden na sesję", gdzie czas życia serwera jest powiązany z czasem życia proxy po stronie klienta i każde proxy „obsługiwane" jest przez jeden serwer, oraz „jeden dla wszystkich", gdzie wszystkie wywołania wykonywane są przez jeden obiekt. Czas życia ma kluczowe znaczenie dla danych przechowywanych w polach obiektu serwera. W modelu „jeden obiekt na wołanie" otrzymujemy bezstanowy model obsługi – zawartość pól nie przeniesie się pomiędzy wołaniami, gdyż kolejne wołanie obsłuży inny obiekt.

Kanał / binding

Usługi sieciowe są w dużej mierze niezależne od kanału komunikacyjnego. Zazwyczaj używa się protokołu HTTP bądź HTTPS, jednak przejście na np. nazwane potoki czy MSMQ nie stanowi problemu i często sprowadza się do zmiany jednej z opcji konfiguracyjnych. Opcja ta nosi nazwę kanału. W angielskiej terminologii jest to binding. Kanał zajmuje się transportem informacji pomiędzy klientem i serwerem i, poza jego konfiguracją, będziemy się nim zajmowali sporadycznie.

Endpoint

Endpoint jest punktem dostępu do usługi. Endpoint to trójka: usługa, binding, adres. Usługę i binding omówiliśmy wcześniej. Adres jest konkretnym adresem dla danego protokołu i zależy od wybranego kanału. Jeżeli, na przykład, zdecydowaliśmy się na protokół (binding) HTTP to adres będzie konkretną nazwą serwera, z którym należy się połączyć (np. localhost:12345, czy www.uslugi.ksr.pl),

Jeden obiekt (serwer) może udostępniać wiele endpointów. Możliwe jest udostępnienie tej samej usługi przez dwa różne kanały, czy nawet przez ten sam kanał, lecz pod różnymi adresami. Z drugiej strony, przez jeden kanał możemy udostępniać różne usługi, o ile znajdują się pod różnymi adresami.

Serwer może udostępniać także specjalny endpoint, określany jako Metadata Exchange (mex). Tym, co różni mex od zwykłego endpointu jest jego przeznaczenie – nie udostępnia on usługi lecz publikuje dane opisujące świadczone usługi. Często sprowadza się do do udostępniania pliku w formacie WSDL.

WSDL

Web Service Description Language jest językiem opisu usług sieciowych, wykorzystującym XML. Opisuje on całość usługi – adres, funkcje, ich parametry, wartości zwracane itp.. Dzięki WSDL, aby wygenerować proxy, wszystko co musimy zrobić, to wskazać środowisku adres pliku WSDL (lub adres końcówki mex serwera) i kliknąć „OK".

SOAP

Simple Object Access Protocol jest specyfikacją, ponownie bazującą na XML, wymiany informacji. Jednym z zastosowań SOAP jest wymiana komunikatów pomiędzy proxy a stubem podczas komunikacji z usługą sieciową. Z jednej strony, elastyczność pozwala na wykorzystanie jednej specyfikacji w komunikacji poprzez różne kanały. Wykorzystanie XML powoduje jednak spory narzut komunikacyjny, a co za tym idzie, spadek wydajności.

2. Windows Communication Foundation

Windows Communication Foundation jest jednym z podejść do realizacji usług sieciowych. Usługa jest tu rozbita jest na kontrakt, czyli interfejs opisujący dostępne metody oraz serwer, czyli klasę implementującą te metody.

Kontrakt jest zwykłym interfejsem w języku C#, posiadającym adnotację ServiceContract. Jest ona wymagana – bez niej interfejs nie zostanie rozpoznany jako kontrakt. Metody interfejsu udostępniamy w usłudze wykorzystując adnotację OperationContract. Wymaga ona przestrzeni nazw System.ServiceModel. Może także być konieczne dodanie do projektu referencji do System.ServiceModel.dll (References -> Add Reference -> Framework). Jeżeli chcemy skorzystać z typów złożonych, czyli klas muszą być one opisane razem z interfejsem i mieć one adnotację DataContract (przestrzeni nazw System.Runtime.Serialization), a ich pola adnotację DataMember. Przykładowy kontrakt może mieć postać:

```
[ServiceContract]
public interface ICalculator {
      [OperationContract]
      int Add(int a, int b);
      [OperationContract]
      int Eval(Data data);
      }

[DataContract]
public class Data {
      [DataMember]
      public int arg1 { get; set; }
      [DataMember]
      public int arg2 { get; set; }
      [DataMember]
      public char op { get; set; }
      }
```

Implementacja kontraktu nie różni się niczym od implementacji zwykłego interfejsu. Tworzymy zwykłą klasę implementującą ten interfejs:

```
public class Calculator: ICalculator {
      public int Add(int a, int b) {
            return a + b;
            }
      public int Eval(Data d) {
            switch(d.op) {
                  case '+': return d.arg1 + d.arg2;
                  ...
                  }
            }
      }
```

Tak jak klasa może implementować kilka interfejsów, tak samo jeden serwer może implementować kilka kontraktów -- wystarczy, że implementuje kontrakty je opisujące.

Tak zaimplementowany serwer musimy udostępnić klientom. Aby to zrobić, pierwszą rzeczą jest wybranie kanału komunikacyjnego, przez który usługa będzie dostępna oraz wybranie adresu powiązanego z tym kanałem.

3. Binding

WCF umożliwia skorzystanie z różnych kanałów komunikacyjnych. Jego zmiana sprowadza się do zmiany konfiguracji serwera i adresu. Wybrane kanały komunikacyjne przedstawia poniższa tabela:

mechanizm	klasa bindingu	schemat adresu przykład	mex endpoint	sesje i interfejsy zwrotne
HTTP	BasicHttpBinding	http://host:port/adres/adres http://localhost:1234/Usluga	CreateMexHttpBinding	NIE
HTTP	WSDualHttpBinding	http://host:port/adres/adres http://localhost:1234/Usluga	CreateMexHttpBinding	TAK
nazwane potoki	NetNamedPipeBinding	net.pipe://host/nazwa_potoku net.pipe://localhost/potok	CreateMexNamedPipeBinding	TAK
TCP	NetTcpBinding	net.tcp://host:port/adres net.tcp://127.0.0.1:999/Usluga	CreateMexTcpBinding	TAK
MSMQ	NetMsmqBinding	net.msmq://host/nazwakolejki net.msmq://localhost/kolejka	brak	NIE

Pełną listę dostępnych bindingów można znaleźć pod adresem http://msdn.microsoft.com/en-us/library/ms730879.aspx.

Skorzystanie z bindingu polega na utworzeniu obiektu odpowiedniej klasy oraz przekazaniu go dalej. Jeżeli potrzeba, możemy taki obiekt przed przekazaniem skonfigurować, ustawiając mu odpowiednie pola, np.:

```
var binding = new NetMSMQBinding();
binding.Durable = true;
binding.ExactlyOnce = false;
binding.Security.Mode = NetMsmqSecurityMode.None;
// tu przekazujemy gdzieś binding
```

4. Self-hosting

Serwer WCF może być udostępniany przez dowolny dowolny program czy usługę. Gdy mamy już kontrakt opisujący usługę oraz serwer ją implementujący, musimy utworzyć hosta usługi, tworząc obiekt typu ServiceHost (z przestrzeni nazw System.ServiceModel). Konstruktor ma dwa argumenty: klasę implementującą usługi oraz tablicę adresów bazowych usługi (tablica obiektów typu Uri), np.:

```
var host = new ServiceHost(typeof(Serwer),
                           new Uri[] {new Uri(„http://localhost:1100”)});
```

Jeżeli nie chcemy podawać adresów bazowych, możemy drugi parametr pominąć. Proszę zwrócić uwagę, że pierwszym parametrem jest tu typ klasy implementującej serwer. W takim przypadku każde wołanie klienta zostanie obsłużone przez osobną, nowo utworzoną instancję serwera.

Po utworzeniu hosta musimy dodać mu endpointy. Endpoint do usługa, binding i adres, musimy więc przekazać te trzy parametry do metody AddServiceEndpoint hosta:

```
host.AddServiceEndpoint(typeof(ICalculator),
                        new NetNamedPipeBinding(),
                        "net.pipe://localhost/wcfpipe");
```

Jeżeli adres rozpoczyna się od nazwy protokołu (tak jak tutaj), będzie adresem bezwzględnym – adres będzie dokładnie taki, jak podamy. W przeciwnym wypadku będzie to adres względem odpowiedniego adresu bazowego podanego w konstruktorze ServiceHost.

Na koniec wołamy na hoście metodę Open i serwer jest gotowy do użycia. Po zakończeniu świadczenia usług należy wywołać metodę Close:

```
host.Open();
// czekamy na zakończenie, np.:
Console.ReadKey();
host.Close();
```

5. Klient

Aby skorzystać z serwisu, musimy znać jego endpoint. Klient musi zatem znać kontrakt, binding oraz adres. Musimy zatem dostarczyć klientowi kontrakt-interfejs, czy to umieszczając go bezpośrednio w kodzie klienta, czy wyodrębniając kontrakt do osobnego assembly (projektu) i dodając do niego referencję zarówno w serwerze jak i kliencie. Drugie rozwiązanie jest bezpieczniejsze – czasem jeden kontrakt opisany w dwóch różnych miejscach może nie być rozpoznany jako ten sam typ co może spowodować problemy z połączeniem. Przeniesienie kontraktu do osobnego assembly powoduje, że serwer i klient korzystają z jednego, wspólnego opisu kontraktu.

Znając endpoint, tworzymy w kliencie fabrykę kanałów i wykorzystujemy ją do utworzenia kanału komunikacyjnego, czyli proxy naszej usługi:

```
using System.ServiceModel;
...

// albo dodana referencja do projektu z ICalculator
[ServiceContract]
public interface ICalculator {
      ...
      }
...
// tworzymy fabrykę
var fact = new ChannelFactory<ICalculator>(
                        new NetNamedPipeBinding(),
                        new EndpointAddress("net.pipe://localhost/wcfpipe"));
// tworzymy proxy
var client = fact.CreateChannel();
// korzystamy z usługi
Console.WriteLine("wynik={0}", client.Add(1, 2));
...
// gdy klient nie jest już potrzebny
((IDisposable)client).Dispose();
// fabrykę możemy zwolnić dopiero wtedy, gdy skończymy korzystać z utworzonych
// przez nią kanałów
fact.Close();
```

Binding i adres możemy także przekazać nieco inaczej. Możemy utworzyć fabrykę przekazując jej tylko binding, wówczas adres musimy przekazać jako parametr do metody CreateChannel. Oczywiście, jeżeli w serwerze dodatkowo konfigurujemy binding, taką samą konfigurację powinniśmy wykonać w kliencie.

6. Metadane i zachowania serwera

Proces tworzenia proxy możemy zautomatyzować. Do tego celu musimy udostępnić metadane opisujące świadczone usługi. Udostępnianie metadanych jest usługą taką jak inne, posiada zatem kontrakt, binding i adres. Jedna usługa z metadanymi opisuje wszystkie świadczone usługi, niezależnie od ich adresów czy bindingów. Jeżeli chcemy, aby nasz serwer udostępniał opis usług, musimy sobie tego zażyczyć. Robimy to ustawiając odpowiednie zachowanie naszego hosta (zanim wywołamy na nim metodę Open; potrzebujemy przestrzeni nazw System.ServiceModel.Description):

```
var b = host.Description.Behaviors.Find<ServiceMetadataBehavior>();
if (b == null) b = new ServiceMetadataBehavior();
host.Description.Behaviors.Add(b);
```

oraz dodając endpoint udostępniający metadane:

```
host.AddServiceEndpoint(ServiceMetadataBehavior.MexContractName,
                        MetadataExchangeBindings.CreateMexNamedPipeBinding(),
                        "net.pipe://localhost/metadane");
```

Jako typ kontraktu podajemy tu ServiceMetadataBehavior.MexContractName zaś binding tworzymy korzystając z metody MetadataExchangeBindings.CreateMexNamedPipeBinding() (w przypadku korzystania z bindingu innego niż NetNamedPipeBinding należy wywołać odpowiednią dla niego metodę, zgodnie z tabelą w rozdziale 3).

Udostępnienie metadanych pozwala nam na wykorzystanie Visual Studio (lub narzędzia svcutil.exe) do generowania proxy. Robimy to dodając do projektu zawierającego klienta referencję do usługi (Project -> Add service reference... lub „Add Service Reference..” z menu kontekstowego po kliknięciu na „References” w Solution Explorer). Okno dialogowe, które pojawi się w rezultacie wybrania tego polecenia pozwala nam na wprowadzenie adresu endpointu usługi z metadanymi. Po kliknięciu „Go” i odczekaniu chwili powinniśmy otrzymać listę udostępnianych usług, z których musimy wybrać jedną – tę, dla której proxy chcemy utworzyć (oczywiście usługa musi być w tym momencie uruchomiona). Przycisk „Discover” powoduje przeszukanie bieżącego solution pod kątem projektów udostępniających usługi sieciowe, ale tylko jeżeli korzystamy z projektu typu WCF Service Application (o tym później). Jeżeli klient i serwer są w tym samym solution, jest to szybsza metoda wskazania usługi, gdyż nie musimy ręcznie wprowadzać adresu.

Pole Namespace pozwala nam określić przestrzeń nazw, w której zostanie umieszczone proxy (zostanie ona zagnieżdżona wewnątrz przestrzeni nazw naszego projektu). Przycisk „Advanced” prowadzi do ustawień zaawansowanych, takich jak sposób obsługi wywołań asynchronicznych. Przycisk „OK” zakończy konfigurację i utworzy proxy. Jeżeli interesuje nas kod proxy jest on dostępny z poziomu narzędzia Object Browser (uruchamianego po dwukrotnym kliknięciu na dodanej referencji), po dwukrotnym kliknięciu na klasie *NazwaUsługi*Client.

Skorzystanie z usługi sprowadza się do utworzenia obiektu proxy (*NazwaUsługi*Client) oraz wywołaniu na nim żądanych metod:

```
var client = new ServiceReference1.CalculatorClient();
// korzystamy z usługi
Console.WriteLine("wynik={0}", client.Add(1, 2));
```

Proxy posiada zaszyty wewnątrz adres i binding usługi (dokładniej, jest on zapisany w pliku App.config). Możemy jednak wymusić na proxy skorzystanie z innych parametrów. Wystarczy, że przekażemy binding i adres usługi podczas tworzenia obiektu klienta.

Należy pamiętać, że opis usługi zawiera tylko informacje o danych. Dla klasy z adnotacją DataContract, po stronie klienta będą widoczne tylko te pola lub właściwości klasy, które zostały oznaczone jako DataMember. W szczególności nie będą widoczne żadne metody czy konstruktor, a właściwości zostaną zamienione na zwykłe pola. Zatem klasa opisana przez DataContract jest po stronie klienta w zasadzie zwykłą strukturą.

svcutil.exe

Jest to narzędzie służące, między innymi, do tworzenia klienta usługi WCF na podstawie metadanych. Wywołanie odpowiadające dodaniu referencji z poziomu Visual Studio ma postać:

```
svcutil.exe [/async] /config:nazwa.exe.config /n:*,namespace adres_metadanych
```

gdzie *adres_metadanych* to adres metadanych na podstawie których ma być wygenerowany klient, *nazwa* to nazwa naszego pliku wykonywalnego zaś *namespace* to przestrzeń nazw podana podczas dodawania Service Reference. Dodanie opcji /async powoduje wygenerowanie metod asynchronicznych. Wynikiem takiego wywołania będzie plik z kodem proxy oraz plik typu Application Configuration File przechowujący dodatkowe ustawienia proxy (o nazwie podanym w parametrze /config). Oba te pliki należy dołączyć do swojego projektu, aby móc skorzystać z wygenerowanego proxy. Uruchomienie svcutil.exe bez parametrów wyświetla pomoc dla tego narzędzia.

7. Wyjątki

WCF umożliwia przekazywanie wyjątków rzucanych przez metody naszej usługi. Jest to trochę bardziej skomplikowane niż zwykłe rzucanie wyjątków, ale taki stan rzeczy ma uzasadnienie – jeżeli chcemy rzucać wyjątki zgodnie ze składnią języka, to runtime musi być w stanie odróżnić wyjątki rzucane celowo (do klienta) od tych, które wynikają z błędów w kodzie.

Wyjątek jest w praktyce strukturą przekazywaną od serwera do klienta, musi być więc opisany jako DataContract. Metody rzucające wyjątki powinny być oznaczone adnotacją FaultContract z podanym typem rzucanego wyjątku. Gdy to zrobimy, możemy już rzucać klientowi wyjątki. Musimy to robić w opakowaniu (aby runtime wiedział, że przeznaczone są one do klienta). Opakowaniem tym jest FaultException. Jest to typ generyczny, parametryzowany typem rzucanego wyjątku i znajduje się w przestrzeni nazw System.ServiceModel. Jego konstruktor przyjmuje dwa parametry – wyjątek oraz napis opisujący przyczynę wyjątku (powód wyjątku, *reason*).

Złapanie wyjątku po stronie klienta nie różni się niczym od łapania klasycznych wyjątków. Należy jednak pamiętać, aby wyjątki łapać w opakowaniu (czyli catch(FaultException<XXX> ...) zamiast catch(XXX ...)). Obiekt FaultException posiada pola Detail zawierające rzucony wyjątek oraz Reason zawierający powód wyjątku w postaci tekstowej i który zawiera drugi argument podany podczas rzucania wyjątku.

Uwaga: rzucenie wyjątku może spowodować komunikat debuggera „FaultExcepion was unhandled by user code". W takim przypadku należy pozwolić programowi działać dalej (zamknąć okno i wydać polecenie „Continue"). Wynika on z nadgorliwości debuggera. Informuje nas, że nie obsłużyliśmy wyjątku, ale o to nam właśnie chodzi – ma go złapać runtime i odpowiednio obsłużyć. Jeżeli komunikat nas irytuje, możemy wyłączyć jego wyświetlanie (checkbox w oknie komunikatu) lub uruchomić serwer bez debuggera, w trybie Release.

W przypadku asynchronicznego wołania metod, wyjątki od serwera będą rzucane przez End*Metoda*.

8. Wołania asynchroniczne

Wywołanie funkcji z usługi sieciowej zazwyczaj zajmuje trochę czasu. Z tego powodu preferowanym sposobem wywołania jest wywołanie asynchroniczne.

Pierwszym, co musimy zrobić, to poprosić o wygenerowanie metod pozwalających na asynchroniczne wołanie. Robimy to w zaawansowanych ustawieniach podczas dodawania referencji, lub wybierając z menu kontekstowego polecenie „Configure Service Reference..." po kliknięciu na referencji (po modyfikacji należy jeszcze wydać polecenie „Update Service Reference" aby przebudować proxy). Interesującą nas opcją jest „Generate asynchronous operations". W przypadku Visual Studio 2012 opcją tą jest „Allow generation of asynchronous operations" z podopcją „Generate asynchronous operations".

Wybranie tej opcji spowoduje wygenerowanie dodatkowych metod w proxy, wołania synchroniczne nadal będą możliwe. Dla każdej metody zostaną wygenerowane dwie dodatkowe, o nazwach (*metoda* jest nazwą metody):

`IAsyncResult Begin`*`metoda`*`(`*`argumenty`*`…, funkcja, parametr)`

Przyjmuje ona takie same argumenty jak *metoda* plus dwa dodatkowe – funkcja zwrotna do wywołania i dodatkowy argument do tej funkcji. Podana funkcja zostanie wywołana w momencie zakończenia wywołania asynchronicznego. Parametr jest dodatkową wartością, którą możemy przekazać do wywołanej funkcji. Oba parametry mogą przyjmować wartość null. Wartością zwróconą z Begin*metoda* jest obiekt typu IAscyncResult, który identyfikuje konkretne wywołanie. Jeżeli wywołamy kilka razy równolegle tę samą metodę, jedynie te obiekty pozwolą nam dopasować otrzymane wyniki do konkretnych wywołań.

`retval` `End`*`metoda`*`(IAsyncResult result)`

Metodę tę należy wywołać, gdy będziemy pewni, że wywołanie asynchroniczne się zakończyło (jeszcze raz: nie aby je zakończyć, tylko kiedy się dowiemy, w jakikolwiek sposób, że wynik jest już gotowy). Jej celem jest odebranie wyniku konkretnego wywołania. Przyjmuje ona jeden argument – IAsyncResult identyfikujący wołanie, którego wynik chcemy otrzymać. Wartością zwracaną jest to, co zwróciłaby oryginalna metoda.

Na przykład, jeżeli nasza metoda to int Dodaj(int, int), do życząc sobie metod asynchronicznych otrzymamy dodatkowe metody BeginDodaj i EndDodaj. Aby asynchronicznie dodać dwie liczby powinniśmy wykonać:

```
client = /* utworzenie proxy */
IAsyncResult res = client.BeginDodaj(3, 4, null, null);
/* oczekiwanie na zakończenie obliczeń */
int wynik = client.EndDodaj(res);
```

Informację o zakończeniu wywołania asynchronicznego możemy otrzymać na kilka sposobów. Pierwszy z nich to skorzystanie z właściwości IsCompleted obiektu IAsyncResult. Jeżeli wołanie się zakończyło, będzie ona miała wartość True, w przeciwnym wypadku będzie to False. Drugi sposób, to skorzystanie z uchwytu oczekiwania. Pole AsyncWaitHandle obiektu IAsyncResult zawiera obiekt typu WaitHandle, na którym możemy wywołać metodę WaitOne, aby poczekać na zakończenie operacji. Możemy także skorzystać ze metod: System.Threading..WaitHandle.WaitAll i System.Threading.WaitHandle.WaitAny, które oczekują na zakończenie

odpowiednio wszystkich i dowolnej spośród podanych w tablicy operacji. W obu podanych metodach nie podajemy funkcji zwrotnej do wywołania Begin*metoda* (nie jest ona potrzebna).

Trzeci sposób korzysta z wywołania zwrotnego. W przedostatnim parametrze Begin*metoda* przekazujemy statyczną metodę, która przyjmuje jeden argument: IAsyncResult. Metoda ta zostanie wywołana, gdy asynchroniczne wołanie się zakończy a w parametrze przekazany jej zostanie obiekt identyfikujący zakończone wołanie. Dodatkowy argument (ostatni parametr Begin*metoda*) będzie widoczny jako wynik wywołania metody AsyncState() na obiekcie IAsyncResult (będzie on typu object, więc należy rzutować go na potrzebny typ). Przykład użycia tego schematu znajduje się poniżej.

```
...
static void Callback(IAsyncResult res)
{
      /* przekazaliśmy klienta w dodatkowym argumencie, bo musimy go mieć,
   żeby wywołać na nim EndAdd */
      var client = (Namespace.UslugaClient)res.AsyncState;
      MessageBox.Show(client.EndDodaj(res).ToString());
}
...
client = /* utworzenie proxy */
IAsyncResult res = client.BeginDodaj(3, 4, Callback, client);
/* proszę się upewnić, że program nie zakończy się zbyt wcześnie – inaczej
   nie będzie szansy na wywołanie Callbacka */
```

Jeżeli zamiast opcji „Generate asynchronous operations" wybierzemy „Generate task-based operations", zamiast par Begin*Metoda* i End*Metoda* otrzymamy asynchroniczną metodę *Metoda*Async korzystającą z zadań (zwraca obiekt typu Task<*typ wartości zwracanej przez metodę*>).

Jeżeli spodziewamy się obsługiwać w serwerze wołania asynchroniczne, warto ustawić mu odpowiedni tryb współbieżności. Robimy to, dodając do klasy serwera adnotację ServiceBehavior z parametrem ConcurrencyMode ustawionym na jedną z wartości:

- Single – wołania obsługiwane są w jednym wątku; następne wywołanie zostanie obsłużone dopiero wtedy, gdy zakończy się poprzednie;
- Reentrant – wołania są obsługiwane przez jeden wątek; gdy serwer oczekuje na odpowiedzi z innych usług, może obsłużyć przychodzące wołania;
- Multiple – wołania mogą być obsługiwane przez różne wątki; należy samodzielnie obsłużyć synchronizację.

9. Wywołania zwrotne

Załóżmy, że udostępniamy usługę wykonującą długotrwałe obliczenia. Chcielibyśmy ponadto, aby użytkownik (klient) mógł śledzić postęp wykonania obliczeń (chociażby w postaci paska postępu). Wykonanie tego dostępnymi dotychczas środkami jest wykonalne, ale uciążliwe. Moglibyśmy na przykład stworzyć metodę WykonajObliczenia(nr_etapu), a kod klienta miałby postać:

```
client = /* utwórz klienta */
for(i = 0; i < 10; i++) {
      pasek.UstawProcent(i * 10)
      client.WykonajObliczenia(i)
      }
pasek.UstawProcent(100);
```

Inny sposób to udostępnienie przez klienta usługi, pozwalającej raportować serwerowi stan postępu:

klient:

```
/* uruchom usługę do
   raportowania postępu */
proxy = /* utwórz klienta */
proxy.WykonajObliczenia();
```

serwer:

```
WykonajObliczenia() {
      k = /* utwórz proxy do usługi
             klienta */
      for(int i = 0; i < 10; i++) {
            k.UstawProcent(i * 10);
            Oblicz(i);
            };
      k.UstawProcent(100);
      }
```

Problemem w tym podejściu jest konfiguracja proxy po stronie serwera. Dla każdego podłączającego się klienta, wystawiona przez niego usługa będzie miała inny adres, musiałby więc podawać go serwerowi. Ponadto klient musi mieć możliwość udostępnienia swojej usługi, co może być problemem za firewallem.

Rozwiązaniem są interfejsy zwrotne. W tym podejściu kontrakt usługi wymaga od klienta implementacji pewnego interfejsu, z którego może korzystać serwer:

klient:

```
handler = /* obiekt spełniający
             interfejs IZwrotny */
proxy = /* utwórz klienta, podając
           obiekt handler */
proxy.WykonajObliczenia()
```

serwer:

```
interface IZwrotny {
      void UstawProcent(int);
            }
[ServiceContract]
...
/* oświadczamy, że IZwrotny jest
   interfejsem zwrotnym */
...
WykonajObliczenia() {
IZwrotny z = /* pobierz interfejs zwrotny */
for(int i = 0; i < 10; i++) {
      z.UstawProcent(i * 10);
      Oblicz(i);
      };
z.UstawProcent(100);
}
```

Rozwiązanie to jest wygodne, ale ma pewne wymagania – nie każdy binding jest w stanie je obsłużyć (zob. np. tabela w rozdziale 3). Po drugie, wywoływane metody muszą być jednokierunkowe. Metoda jednokierunkowa to taka, która nic nie zwraca (void) oraz adnotacja OperationContract ma ustawiony parametr IsOneWay na true. Dzięki temu wołający metodę nie musi czekać na jej zakończenie.

Podsumowując, użycie interfejsu zwrotnego mogłoby mieć postać:

```
/* strona serwera */
/* interfejs zwrotny nie jest kontraktem,
   nie ma więc adnotacji ServiceContract */
interface IZwrotny {
      /* nie ma ServiceContract, ale OperationContract musi być;
         serwer nie czeka na odpowiedź klienta */
      [OperationContract(IsOneWay = true)]
      void UstawProcent(int);
      }
...
/* podanie typu interfejsu zwrotnego */
[ServiceContract(CallbackContract=typeof(IZwrotny))]
interface IService1 {
      [OperationContract(IsOneWay=True)]
      void WykonajObliczenia();
      }
...
```

```
class Service1: IService1 {
      ...
      public void WykonajObliczenia() {
            var zwr = OperationContext.Current.GetCallbackChannel<IZwrotny>();
            ...
            zwr.UstawProcent(100);
            ...
            }
      }
```

Aby otrzymać interfejs zwrotny po stronie serwera, musimy wywołać metodę GetCallbackChannel na polu Current obiektu OperationContext. Jest to metoda generyczna, której argumentem jest typ interfejsu, który chcemy pobrać. Należy pamiętać, że z jedną usługą może być związany tylko jeden interfejs zwrotny.

Klient z kolei musi dostarczyć implementacji interfejsu. Jego opis jest umieszczony w metadanych zatem dodanie referencji do usługi da nam także opis interfejsu. Trzeba jednak pamiętać, że nazwa interfejsu zostanie zmieniona na *nazwakontraktu*Callback, czyli zamiast naszego IZwrotny będziemy mieli np. IService1Callback. Interfejs ten musimy zaimplementować w jakiejś klasie i obiekt tej klasy przekazać do proxy (opakowany dodatkowo w obiekt InstanceContext) w konstruktorze. Obiekt ten będzie obsługiwał wszystkie wołania zwrotne serwera.

```
class Handler: ServiceReference1.IService1Callback {
      public void UstawProcent(int pct) {
            ...
            }
      }
...
var c = new ServiceReference1.Service1Client(new InstanceContext(new Handler()));
```

Jeżeli nie korzystamy z metadanych, do utworzenia kanału powinniśmy skorzystać z DuplexChannelFactory zamiast ChannelFactory. Klasa ta przyjmuje dodatkowy parametr (poza adresem i bindingiem) w którym należy przekazać, opakowany w InstanceContext, obiekt obsługujący wołania zwrotne (podobnie, jak robimy to w przypadku klienta tworzonego z metadanych).

10. Czas życia serwera

Jeżeli nie zażądamy inaczej, każde wywołanie metody będzie obsługiwane przez nowy obiekt serwera. Z tego powodu podajemy klasę a nie obiekt tworząc ServiceHost. Jeżeli jednak takie zachowanie nam nie odpowiada, możemy je zmienić. Robimy to dodając do **klasy serwera (nie interfejsu)** adnotację ServiceBehavior z parametrem InstanceContextMode. Możliwe wartości to:

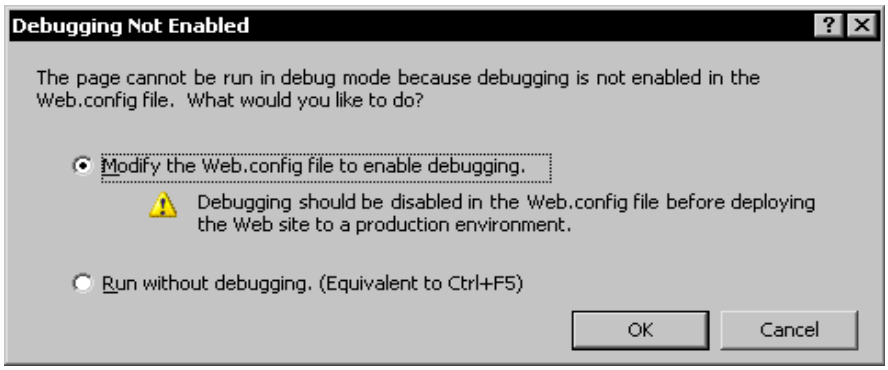
InstanceContextMode.PerCall	- nowy obiekt na każde wołanie metody
InstanceContextMode.PerSession	- nowy obiekt na każde proxy; w tym przypadku binding musi obsługiwać sesje oraz musimy do adnotacji ServiceContract interfesju dodać parametr: SessionMode=SessionMode.Allowed lub SessionMode=SessionMode.Required
InstanceContextMode.Single	- jeden obiekt do wszystkich; w tym przypadku możemy w pierwszym parametrze konstruktora ServiceHost podać nie typ lecz obiekt, który ma obsługiwać wołania

Z tak uruchomionego serwera możemy korzystać w ten sam sposób, w jaki korzystaliśmy z serwera przygotowanego przez Visual Studio. Jedyną różnicą jest to, że przycisk „Discover” podczas dodawania referencji nie znajdzie naszego serwera, będziemy musieli więc uruchomić go ręcznie i ręcznie wpisać adres, pod którym dostępny jest WSDL.

Na czas tworzenia i debuggowania serwera warto jest dodać do klasy implementującej serwer adnotację ServiceBehavior z parametrem IncludeExceptionDetailInFaults ustawioną na True. Powoduje to dołączenie szczegółowego opisu błędu w różnych wyjątkowych sytuacjach (np. podczas problemów z utworzeniem proxy).

11. WCF Service Application

Serwery WCF mogą być hostowane przez serwer IIS. Serwery takie tworzymy jako projekty typu WCF Service Application. Nowo utworzony projekt jest gotowy do uruchomienia i zawiera prostą usługę sieciową. Za pierwszym razem, gdy uruchomimy go to wybierając opcję Run (F5) możemy zobaczyć okno dialogowe:



Jest to pytanie, czy środowisko ma zmodyfikować plik Web.config w naszym projekcie, aby umożliwić debuggowanie. Konsekwencje są podobne do zwykłej kompilacji w trybie Debug – możemy umieszczać w kodzie pułapki i w pełni korzystać z debuggera, lecz wydajność naszej aplikacji jest przez to ograniczona. W każdej chwili możemy ręcznie zmienić to ustawienie, modyfikując parametr węzła system.web -> compilation o nazwie debug w pliku Web.config:

```
<system.web>
      ...
      <compilation debug="true">
```

Uruchomienie projektu spowoduje wystartowanie serwera wbudowanego w Visual Studio, a ten zacznie udostępniać naszą usługę. Zostanie także uruchomiona przeglądarka, w której zostanie otworzona strona testowa usługi (jeżeli tak się nie stanie, zawsze możemy uruchomić ją ręcznie i wpisać adres usługi w pasku adresu, należy pamiętać również o podaniu portu). Jeżeli serwer nasłuchuje na porcie 1229, adres będzie miał postać http://localhost:1229/Service1.svc. Port może zmienić się z każdym uruchomieniem serwera, możemy jednak ustawić jego wartość w opcjach projektu w zakładce Web, w sekcji Servers wybierając „Use Visual Studio Development Server" i w Specific Port podając numer portu. Pominięcie Service1.svc w adresie wyświetli nam listę plików w naszym projekcie. Wybranie Service1.svc przeniesie nas do strony testowej. Do testowania usług WCF dostępne jest narzędzie wcftestclient.

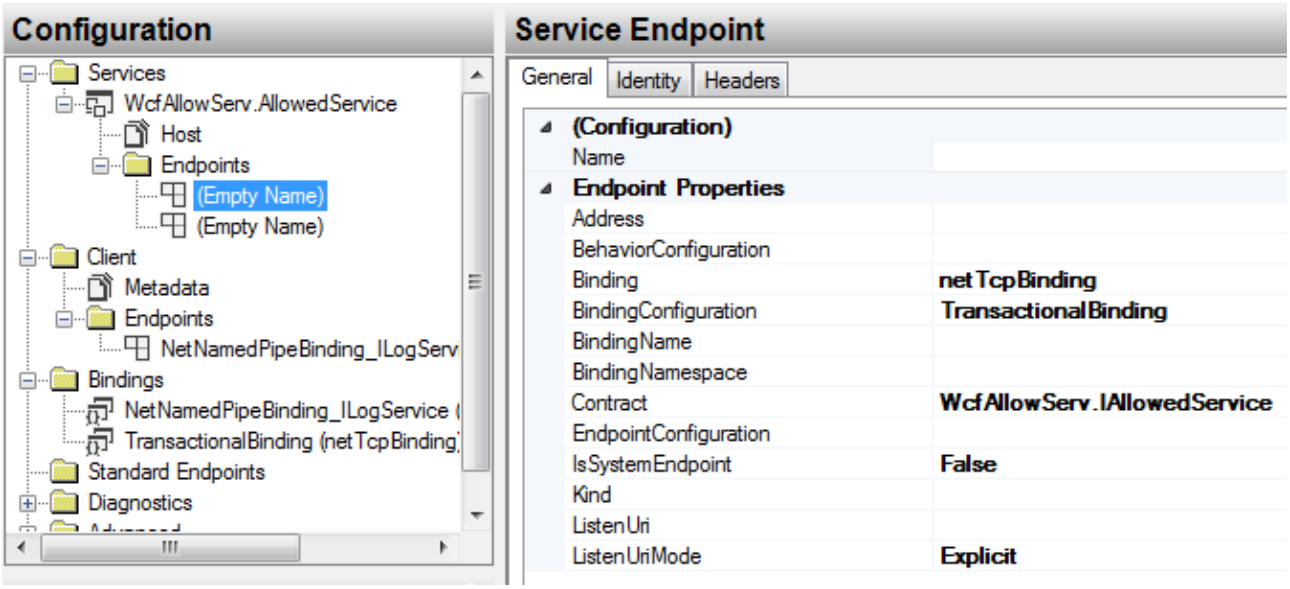
Aplikacja tego typu domyślnie udostępnia metadane, zatem utworzenie proxy wymaga tylko kilku kliknięć.

WCFTestClient

Uruchomienie tego narzędzia wymaga wydania polecenia wcftestclient.exe (w „Visual Studio Command Prompt", aby ścieżki były poprawnie ustawione). Pierwsze, co powinniśmy zrobić po jego uruchomieniu, to dodać usługę (menu File, polecenie Add service) wprowadzając ten sam adres, który pojawił się w przeglądarce (może być to także adres pliku WSDL). Następnie możemy wywoływać nasze metody, oglądać przesyłane komunikaty itp. Dzięki opisaniu typów złożonych w kontrakcie nie jesteśmy ograniczeni do przekazywania typów prostych jako argumentów wywołania, możemy przekazać także struktury, wypełniając każde ich pole. WCFTestClient nie jest ograniczony do testowania serwisów hostowanych przez IIS. Równie dobrze możemy testować nim serwisy udostępnione poprzez ServiceHost.

12. Konfiguracja i SvcConfigEditor

Informacje opisujące usługę umieszczone są w pliku `Web.config` (projekt WCF Service Application – hostowany w IIS/WAS) albo `App.config` (projekt WCF Service Library – hostowany w WcfSvcHost.exe/hoście własnym). Jest to plik w formacie XML, można więc edytować go ręcznie. Wygodniejsze jest jednak skorzystanie z gotowego edytora konfiguracji: SVCConfigEditor. Po kliknięciu prawym przyciskiem myszy na pliku Web.config w SolutionExplorer wybieramy „Open with..." następnie Add i wskazujemy edytor: C:\Program Files\Microsoft SDKs\Windows\v7.0A\bin\NETFX 4.0 Tools\SvcConfigEditor.exe W przypadku Visual Studio 2012 i wyższych dostęp do edytora otrzymujemy wybierając opcję „Edit WCF Configuration" lub „WCF Service Configuration Editor" (menu Tools).



Konfiguracja serwisu opisana jest w sekcji `<system.serviceModel>` pliku konfiguracyjnego. Po utworzeniu nowego projektu do sekcji konfiguracyjnej domyślnie zostają dodane podsekcje `services` i `behaviors`. W **`services`** opisane są udostępniane usługi. Domyślnie projekt udostępnia jedną usługę dla której zdefiniowana jest nazwa (wartość atrybutu `name`), adres bazowy (wartość atrybutu `baseAddress` elementu `host → baseAddresses → add`[1]) oraz dwa punkty końcowe (**`endpoint`**). Pierwszy to nasza usługa a drugi to opis danych usługi (mex). Dla każdego z nich zdefiniowana jest trójka atrybutów ABC (**`address, binding, contract`**). W sekcji `behaviors → serviceBehaviors → behavior` znajdziemy atrybuty **`httpGetEnabled`** i **`httpsGetEnabled`** elementu **`serviceMetadata`**. Włączają one udostępnianie metadanych za pomocą metody GET protokołu http oraz https i powinny być ustawione na **`false`** w przypadku wykorzystania innego protokołu niż wymieniony np. net.tcp lub net.pipe.

Jeśli do projektu dodamy web referencję do innej usługi to w sekcji `<system.serviceModel>` dodana zostanie w podsekcji **`client → endpoint`** informacja o punkcie końcowym dodanej usługi oraz podsekcja **`binding`** zawierające dostępne konfiguracje wiązań. Po dokonaniu zmian w konfiguracji usługi najprawdopodobniej będzie wymagane uaktualnienie tych zmian w każdym z klientów korzystających z edytowanej usługi. Zamiast dokonywać ręcznej edycji plików konfiguracyjnych każdego z klientów powinno wystarczyć zaktualizowanie bądź usunięcie i dodanie na nowo Web Referencji w każdym z nich.

Ponieważ **`System.Configuration`** nie obsługuje plików konfiguracyjnych bibliotek, konfigurując deklaratywnie serwis hostowany w hoście własnym należy dołączyć zawartość pliku konfiguracyjnego usługi do pliku konfiguracyjnego hosta. Zostało to opisane w komentarzu w pliku konfiguracyjnym: `<!-- When deploying the service library project, the content of the config file must be added to the host's app.config file. System.Configuration does not support config files for libraries. -->`.

13. REST i Web Programming Model

WCF umożliwia udostępnienie usług za pośrednictwem zwykłych zapytań HTTP. Nie korzystamy tu z SOAP do opakowania wiadomości, a wszystkie dane przekazywane są w treści zapytania zgodnie z protokołem HTTP. Dzięki temu możemy niewielkim nakładem pracy skorzystać z naszej usługi z poziomu kodu JavaScript umieszczonego w witrynie internetowej.

Serwer

Aby skorzystać z tej metody udostępniania serwisu musimy skorzystać z bindingu WebHttpBinding oraz dodać do endpointa zachowanie o nazwie WebHttpBehavior():

[1] W przypadku hostowania usługi w IIS możemy dodać kilka adresów bazowych (kilka elementów add) nie jest to jednak domyślne zachowanie i wymaga ustawienia wartości atrybutu `multipleSiteBindingsEnabled` na `true`. W przypadku innego hostowania dodawanie większej liczby adresów zakończy się wyjątkiem.

```
var sh = new ServiceHost(typeof(Server), new Uri("http://localhost"));
var ep = sh.AddServiceEndpoint(typeof(IUsluga), new WebHttpBinding(), "Usluga");
ep.Behaviors.Add(new WebHttpBehavior());
```

Alternatywą jest skorzystanie z obiektu WebServiceHost, który automatycznie tworzy odpowiedni endpoint i dodaje do niego zachowanie (o ile serwer implementuje tylko jedną usługę):

```
var sh = new WebServiceHost(typeof(Server), new Uri("http://localhost/Usluga"));
```

Musimy skorzystać, czyli dodać referencję, z assembly System.ServiceModel.Web. Aby assembly stało się dostępne, musimy zmienić „Target Framework" w ustawieniach projektu (zakładka „Application") na „.Net Framework 4.0" (lub wyższa wersja; nie powinno to być „Client Profile"). Proszę zwrócić uwagę, że nie dodajemy tu żadnych endpointów. Oczywiście możemy także dokonać odpowiednich zmian w pliku konfiguracyjnym Web.config – należy dodać usługę z endpointem i zmienić mu binding na webHttpBinding. Następnie należy dodać nowe zachowanie endpointu (Advanced -> Endpoint Behaviors) i dodać do niego webHttp. Następnie to zachowanie należy przypisać do naszego endpointu (Services -> Endpoints -> ... -> BehaviorConfiguration).

Drugą czynnością do wykonania jest rozszerzenie opisu kontraktu. Ponieważ w zapytaniu HTTP nie ma miejsca na nazwę wywoływanej metody, musimy określić w jakie adresy zapytania będą odpowiadały jakim metodom. Robimy to wykorzystując adnotacje WebGet i WebInvoke, które dodajemy do poszczególnych metod. Adnotacja WebGet jest szczególnym przypadkiem adnotacji WebInvoke – określa, że dana metoda może być wywoływana metodą HTTP GET, podczas gdy WebInvoke zezwala na dowolne metody wołania. Obie adnotacje maja parametr UriTemplate który opisuje szablon adresu. Metoda będzie obsługiwała zapytania z adresami zgodnymi z szablonem. Fragmenty szablonu w postaci {*nazwa*} (nazwa otoczona przez nawiasy klamrowe) traktowane są jako parametry. Część adresu, która odpowiada parametrowi w szablonie zostanie przekazana jako wartość argumentu o nazwie *nazwa* do metody.

```
...
[OperationContract]
[WebGet(UriTemplate = "dodaj/{a}/{b}")]
int Add(string a, string b);
...
int Add(string a, string b) {
      return Int32.Parse(a) + Int32.Parse(b);
      }
```

Zaimplementowanie takiej metody w usłudze dostępnej pod adresem http://localhost/Usluga spowoduje, że zapytanie GET o zasób http://localhost/Usluga/dodaj/111/12 zwróci tekst <int>123</int>. Jak widzimy, 111 z adresu zostało przekazane jako parametr a (zgodnie z szablonem) zaś 12 trafiło do parametru b. Należy pamiętać, aby parametry przechwytywane z szablonu miały typ string (inaczej zostanie rzucony wyjątek System.InvalidOperationException).

Jeżeli nie określimy inaczej, wartość zwrócona zostanie zakodowana w XML. Możemy to zmienić, ustawiając parametr adnotacji o nazwie ResponseFormat na WebMessageFormat.Json. Adnotacja WebInvoke posiada także parametr Method, którym możemy ustawić obsługiwaną metodę HTTP (GET, PUT, POST, itp.). Domyślną metodą dla WebInvoke jest POST.

Niestety, bez dodatkowego nakładu pracy WCF obsługuje tylko formaty Json i XML, zatem przetestowanie zapytań innych niż GET z wykorzystaniem prostego formularza będzie utrudnione. Na szczęście mamy dostęp do „surowej" zawartości treści zapytania. Jeżeli ostatnim parametrem metody będzie zmienna typu System.IO.Stream, to w strumieniu tym otrzymamy treść zapytania.

```
...
[OperationContract]
[WebInvoke(UriTemplate = "testformularza")]
int Formularz(System.IO.Stream s);
...
int Formularz(System.IO.Stream s) {
      byte[] d = new byte[s.Length];
      data.Read(d, 0, (int)s.Length);
      string s2 = ASCIIEncoding.ASCII.GetString(d);
      Console.WriteLine("zawartosc formularza = {0}", s2);
      return 42;
      }
...
```

```
<html>
<body>
<form action="http://localhost/Usluga/testformularza" method="POST">
pole1:<input type="text" name="pole1"><br>
pole2:<input type="text" name="pole2"><br>
<input type="submit">
</form>
</body>
</html>
```

Uwaga: jeżeli chcemy skorzystać z AJAX do przetestowania naszej usługi musimy uważać na zabezpieczenia przeglądarki związane z XSS. W skrócie, przeglądarka nie pozwoli nam wykonać zapytania AJAX do domeny innej niż ta, z której pochodzi dokument wykonujący zapytanie (nawet z file:/// do http://localhost:...). Można to obejść, wykorzystując serwer WCF jako prosty serwer WWW, dzięki czemu dokument testowy i cel zapytania znajdzie się w tej samej domenie:

```
using System.Xml;
...
[ServiceContract]
public interface IHttpServer {
      [OperationContract,
       WebGet(UriTemplate = "index.html"),
       XmlSerializerFormat]
      XmlDocument Index();
      }
...
public XmlDocument Index() {
      var d = new XmlDocument();
      d.XmlResolver = null;
      d.LoadXml("<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 Strict//EN\" "+
                "\"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd\">"+
                "<html xmlns=\"http://www.w3.org/1999/xhtml\" xml:lang=\"pl\" "+
                  "lang=\"pl\"><head><title>Strona testowa</title>"+
                "<meta http-equiv=\"Content-Type\" content=\"text/html; "+
                "charset=utf-8\" />"+
                "</head><body><p>Strona testowa.</p></body></html>");
      return d;
      }
```

Drugą metodą, która pozwoli nam na zbudowanie prostego serwera WWW jest skorzystanie z obiektu WebOperationContext.current. Daje on nam dostęp do informacji o zapytaniu, a także umożliwia modyfikowanie parametrów odpowiedzi. Pole IncomingRequest opisuje przychodzące zapytanie i posiada między innymi pola ContentEncoding, ContentLength i Headers pozwalające na łatwy dostęp do nagłówków. Z kolei w polu OutgoingResponse możemy ustawiać te parametry. Powinniśmy także zwrócić System.IO.Stream, jeżeli nie chcemy, aby zwrócony napis został opakowany w znaczniki XML:

```
public System.IO.Stream Index() {
      var ctx = WebOperationContext.Current.IncomingRequest;
      foreach (var k in ctx.Headers) {
            Console.WriteLine("{0}: {1}", k, ctx.Headers.Get(k.ToString()));
            }
      OutgoingWebResponseContext context =
                                    WebOperationContext.Current.OutgoingResponse;
      context.ContentType = "text/html";
      return new System.IO.MemoryStream(
            ASCIIEncoding.Default.GetBytes("<html><body>tekst</body></html>"));
      }
```

Klient

Nic nie stoi na przeszkodzie, aby skorzystać z naszego serwisu z poziomu kodu w .Net. Jedynym problemem jest to, że nie będziemy w stanie wygenerować automatycznie kodu proxy, dodając Service Reference (dodanie

referencji może sprawiać wrażenie, że wszystko się udało, ale utworzone w ten sposób proxy nie będzie działało poprawnie; odpowiedź na pytanie dlaczego tak się dzieje można znaleźć w [1]). „Ręczne" utworzenie proxy nie różni się wiele od tworzenia go dla zwykłych usług – jedyna różnica to konieczność dodania zachoania WebHttpBehavior do endpointu:

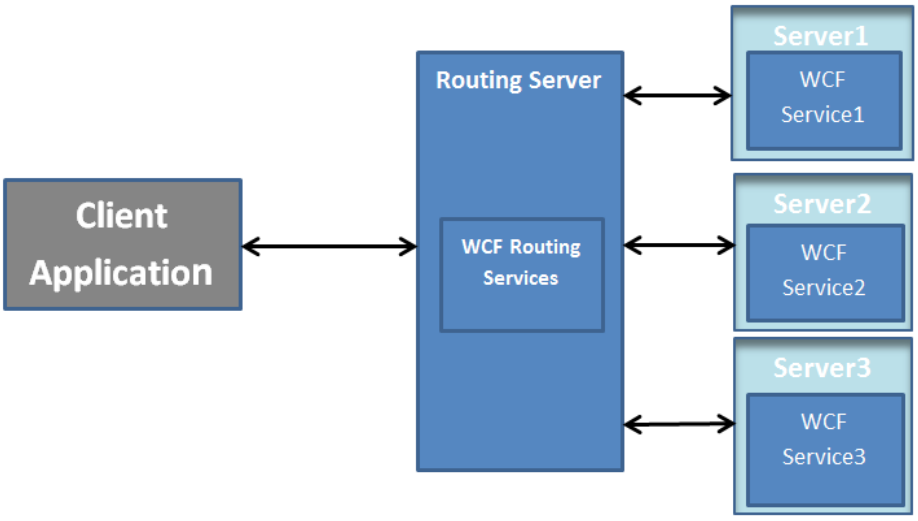
```
using System.ServiceModel;
using System.ServiceModel.Web;
...

[ServiceContract]
public interface IUsluga {    // musimy "ręcznie" opisać usługę
      [OperationContract, WebGet(UriTemplate="metoda/{s}")]
      string metoda(string s);
      }
...
var f = new ChannelFactory<IUsluga>(new WebHttpBinding(),
                                    new EndpointAddress("http://localhost:1234"));
f.Endpoint.Behaviors.Add(new WebHttpBehavior());
var c = f.CreateChannel();
Console.WriteLine(c.metoda("test"));
((IDisposable)c).Dispose();
```

Lektura uzupełniająca:

[1] http://blogs.msdn.com/b/carlosfigueira/archive/2012/03/26/mixing-add-service-reference-and-wcf-web-http-a-k-a-rest-endpoint-does-not-work.aspx

14. Routing

Routing jest usługą WCF przekazującą wołania do innych usług. Zastosowaniem może być m.in. logowanie, priorytetyzacja, wyrównywanie obciążenia czy przesyłanie wołań w zależności od dostępności serwerów. Tworzy on dodatkową, pośrednią warstwę na drodze wołania metody.



Źródło: http://www.codeproject.com/Articles/423064/Getting-Started-with-WCF-4-0-Routing-Service

Stworzenie prostego routera, przekazującego wiadomości, przedstawia poniższy kod:

```
using System.ServiceModel.Routing;
using System.ServiceModel.Dispatcher;
...
var routeTo = "http://localhost:1234/Usluga";
var routerAddr = "http://localhost:1000/Router";
var h = new ServiceHost(typeof(RoutingService));
h.AddServiceEndpoint(typeof(IRequestReplyRouter),
                     new BasicHttpBinding(), routerAddr);
var rc = new RoutingConfiguration();
var contract = ContractDescription.GetContract(typeof(IRequestReplyRouter));
var client = new ServiceEndpoint(contract, new BasicHttpBinding(),
                                 new EndpointAddress(routeTo));
var lst = new List<ServiceEndpoint>();
lst.Add(client);
rc.FilterTable.Add(new MatchAllMessageFilter(), lst);
h.Description.Behaviors.Add(new RoutingBehavior(rc));
```

Jak zwykle, tworzymy obiekt ServiceHost (pomijając tablicę adresów bazowych). W naszym przypadku obiektem obsługującym żądania będzie gotowy obiekt RoutingService. Następnie dodajemy endpoint dla usługi IRequestReplyRouter. Usługa ta obsługuje wołania typu żądanie-odpowiedź (np. funkcje zwracające wartości). Gotowe usługi routingowe to:

ISimplexDatagramRouter - obsługuje sesje, tylko wołania jednokierunkowe
ISimplexSessionRouter - wymaga sesji, tylko wołania jednokierunkowe
IRequestReplyRouter - obsługuje sesje, tylko wołania typu żądanie-odpowiedź
IDuplexSessionRouter - wymaga sesji i bindingu obsługującego wołania zwrotne, obsługuje zarówno wołania jednokierunkowe jak i żądanie-odpowiedź

Ustawiamy także binding oraz adres. Jest to adres, na którym "nasłuchuje" router – z tym adresem będzie łączył się klient. W kolejnych krokach opisujemy, gdzie mają trafiać odebrane przez router wołania. Polega to na zdefiniowaniu par filtr-cel w tablicy filtrów. Filtr jest obiektem dopasowującym wołania, cel jest endpointerm, do którego mają one być przekazane. W powyższym przykładzie wszystkie wiadomości (MatchAllMessageFilter) przekazywane mają być do endpointu opisanego przez zmienną client. Jeżeli więcej niż jeden wpis w tablicy filtrów dopasowuje dane wołanie i wołanie jest jednokierunkowe, otrzymamy multicast – wołanie zostanie przesłane do wszystkich endpointów. Listę dostępnych filtrów można znaleźć pod adresem [2]

Endpointy docelowe podajemy w formie listy. Jeżeli lista zawiera więcej niż jeden endpoint, kolejne stają się endpointami zapasowymi.. Jeżeli nie powiedzie się przesłanie wołania do pierwszego endpointu z listy, router spróbuje przesłać je do kolejnych (zgodnie z porządkiem na liście). Szczegóły zachowania w przypadku innych błędów są opisane na stronie [1].

Na koniec dodajemy konfigurację do zachowań serwera i jest on gotowy do uruchomienia. Działanie routera można w każdej chwili zmienić, modyfikując jego konfigurację. Aby to zrobić, należy stworzyć obiekt zawierający nową konfigurację (nie mamy możliwości zmodyfikowania poprzedniej – musimy stworzyć nową od zera) a następnie wywołać:

```
var cfg = new RoutingConfiguration();
...
host.Extensions.Find<RoutingExtension>().ApplyConfiguration(cfg);
```

Nowa konfiguracja zacznie obowiązywać dla wszystkich nowych wołań. Wołania w trakcie obsługi zostaną zrealizowane zgodnie z poprzednią konfiguracją.

Klient w naszym przypadku nie będzie korzystał z opisu usługi w metadanych (nie będziemy dodawać referencji do usługi). Aby z nich skorzystać potrzebowalibyśmy nie tylko routera dla usługi udostępniania metadanych, co nie jest trudne do zrobienia, lecz także musielibyśmy ingerować w zawartość opisu usługi – chcemy aby klient kontaktował się z routerem a nie z serwerem docelowym, a to jego adres znalazłby się w metadanych. Na szczęście pominięcie automatycznego tworzenia proxy nie wymaga wiele wysiłku i robiliśmy to już kilka razy.

Lektura uzupełniająca:
[1] http://msdn.microsoft.com/en-us/library/ee517422.aspx
[2] http://msdn.microsoft.com/en-us/library/ee517424.aspx.
[3] http://msdn.microsoft.com/en-us/magazine/cc500646.aspx
[4] http://msdn.microsoft.com/en-us/magazine/cc546553.aspx
[5] http://www.codeproject.com/Articles/423064/Getting-Started-with-WCF-4-0-Routing-Service

15. Publikacja i wyszukiwanie usług

W WCF dostępna jest infrastruktura pozwalająca na wyszukiwanie danych usług – jeżeli znamy binding i kontrakt i nie znamy adresu usługi a serwer umożliwia wyszukiwanie, jesteśmy w stanie zapytać serwer o jego adres. Istnieją dwa modele odpytywania: ad-hoc i zarządzany (z centralnym serwerem). Wyszukiwanie w modelu ad-hoc polega na wysyłaniu zapytań rozgłoszeniowych przez klienta, na które odpowiada serwer.

Konfiguracja serwera do pracy w modelu ad-hoc wymaga dodania zachowania ServiceDiscoveryBehavior:

```
using System.ServiceModel.Discovery;
...
host.Description.Behaviors.Add(new ServiceDiscoveryBehavior());
host.AddServiceEndpoint(new UdpDiscoveryEndpoint("soap.udp://localhost:54321"));
```

Adres będący parametrem UdpDiscoveryEndpoint jest adresem, na którym serwer będzie oczekiwał na zapytania klientów. Możemy pominąć ten parametr, wówczas klasa skorzysta z domyślnego adresu.

Po stronie klienta musimy wykonać nieco więcej pracy. Tworzymy najpierw klienta wyszukiwania, podając mu adres rozgłoszeniowy, na jaki ma wysłać zapytanie. Możemy pominąć ten adres (i skorzystać z domyślnego). Zapytanie zwraca listę znalezionych endpointów, w której wyszukujemy implementujące kontrakt IUsluga. Jeżeli lista jest niepusta, tworzymy proxy dla adresu pierwszego na liście i korzystamy z niego.

```
using System.ServiceModel.Discovery;
...
DiscoveryClient discoveryClient =
      new DiscoveryClient(
                  new UdpDiscoveryEndpoint("soap.udp://10.255.255.255:54321"));

System.Collections.ObjectModel.Collection<EndpointDiscoveryMetadata> lst =
            discoveryClient.Find(new FindCriteria(typeof(IUsluga))).Endpoints;
discoveryClient.Close();
if(lst.Count > 0) {
      var addr = lst[0].Address; // łączymy się z pierwszym znalezionym
      var proxy = ChannelFactory<IUSluga>.CreateChannel(new BasicHttpBinding(),
                                                        addr);
      proxy.metoda(...);
      ((IDisposable)c).Dispose();
      }
```

Więcej informacji można znaleźć pod adresem [1].

Lektura uzupełniająca:
[1] https://msdn.microsoft.com/en-us/library/dd456782%28v=vs.110%29.aspx

16. Transakcje

Zadaniem systemu transakcji jest utrzymanie systemu w spójnym stanie. Powinniśmy tak projektować system, aby przejście z jednego stanu spójnego systemu, nazwijmy go A do innego stanu spójnego B wykonywać w transakcji. Daje nam to gwarancję, że albo wszystkie operacje ujęte w transakcji zostaną poprawnie wykonane i system przejdzie do stanu B albo w razie niepowodzenia przynajmniej jednej operacji wszystkie dotychczasowe zmiany zostaną wycofane i system wróci do stanu A (sprzed transakcji). WCF pozwala na tajne/deklaratywne oraz jawne/imperatywne zarządzanie przepływem transakcji. Jeśli tylko możemy powinniśmy korzystać z tego pierwszego sposobu. Pozwala on na uzyskanie odmiennych trybów przepływu transakcji w zależności od ustawień wiązania, kontraktu i zachowania.

Rodzaje wiązań które poznaliśmy wcześniej można podzielić na wiązania świadome transakcji (WSDualHttpBinding, NetTcpBinding, NetNamedPipeBinding) oraz nieświadome (BasicHttpBinding, NetMsmqBinding). Tylko te pierwsze są w stanie rozpropagować transakcję, jednak nie robią tego domyślnie. W tym celu wymagają zmiany wartości właściwości `TransactionFlow` wiązania na `true`. Możemy to zrobić w kodzie:

```
WSDualHttpBinding wsBinding = new WSDualHttpBinding();
wsBinding.TransactionFlow = true;
```

lub w pliku konfiguracyjnycm (App.config):

```
<wsDualHttpBinding>
      <binding name="TransactionalBinding" transactionFlow="true" />
</wsDualHttpBinding>
```

W kontrakcie na poziomie każdej z metod dysponujemy atrybutem `TransactionFlow`:

```
[OperationContract]
[TransactionFlow(TransactionFlowOption.Allowed)]
```

Może on przyjąć jedną z trzech wartości:

- `NotAllowed` (domyślna) – zignorowanie transakcji klienta,
- `Allowed` – przepływ transakcji klienta do usługi o ile klient takową dysponuje i wiązanie na to pozwala,
- `Mandatory` – wymuszenie przepływu transakcji klienta, co pociąga za sobą istnienie transakcji w kliencie oraz zastosowanie wiązania świadomego transakcji z poprawnie skonfigurowaną właściwością `TransactionFlow`.

Uwaga: w przypadku wybrania opcji innej niż domyślna (NotAllowed) nie możemy wykorzystać wołań jednokierunkowych ponieważ usługa musi posiadać możliwość poinformowania klienta o przerwaniu transakcji, jeśli takie zdarzenie nastąpi.

W celu uzyskania dostępu do otoczenia transakcji musimy zmienić wartość właściwości zachowania operacji `TransactionScopeRequired` na `true`. Pozwoli to na propagację otoczenia transakcji z klienta do usługi podczas przepływu transakcji.

```
[OperationBehavior(TransactionScopeRequired = true)]
```

Odpowiednio konfigurując trzy wyżej wymienione ustawienia (wiązania, kontraktu i zachowania) możemy uzyskać cztery tryby propagacji transakcji:

- tryb „klient/usługa”
  `TransactionFlow = true,`
  `TransactionFlowOption.Allowed,`
  `TransactionScopeRequired = true`
  Jeśli klient posiada własną transakcję usługa dołączy do tej transakcji w przeciwnym przypadku usługa utworzy własną transakcję do której dostęp otrzyma klient. Jest to rozwiązanie optymalne ze względu na spójność systemu (klient i usługa nigdy nie będą dysponować osobnymi transakcjami, z których jedna mogła by się powieść a druga została by odwołana), a zarazem wprowadzające najmniejsze zależności usługi względem klienta. Podczas projektowania usługi jej zachowanie nie powinno być uzależnione od tego czy transakcja rozpoczęła się w kliencie czy w usłudze. Minusem jest wysoka podatność na zakleszczenia (transakcje klienta i usługi mogą rywalizować o dostęp do tych samych zasobów).

- tryb „klient”
  `TransactionFlow = true,`
  `TransactionFlowOption.Mandatory,`
  `TransactionScopeRequired = true`
  Jeśli usługa musi skorzystać z transakcji klienta ten tryb wydaj się być najbardziej odpowiednim. Charakteryzuje się największa spójnością (działanie klienta i usługi zawsze będzie uruchamiane w ramach jednej transakcji) oraz brakiem możliwości zakleszczenia (brak innych transakcji konkurujących o dostęp do tych samych zasobów i blokad).

- tryb „usługa”
  ```
  TransactionFlow = false,
  TransactionFlowOption.NotAllowed,
  TransactionScopeRequired = true
  ```
  Zapewnia, że usługa zawsze będzie dysponować własną transakcją – niezależnie od rodzaju klienta. Może być stosowana gdy zależy nam aby wykonać pewne czynności bez względu na przebieg transakcji po stronie klienta. Przykładowo może to być usługa pozwalająca na raportowanie stanu innych usług. W tym celu wymagana jest osobna transakcja ponieważ raportowanie w ramach odwołanej transakcji spowodowało by również odwołanie tego raportu. Ten tryb powinien być stosowany w sytuacji gdy prawdopodobieństwo powodzenia transakcji usługi będzie dużo wyższe od transakcji klienta. To podejście jest niewłaściwe jeśli sytuacja w której jedna z transakcji (klienta albo usługi) zostanie zatwierdzona a druga odwołana spowoduje rozspójnienie systemu.

- tryb „brak”
  ```
  TransactionFlow = false,
  TransactionFlowOption.NotAllowed,
  TransactionScopeRequired = false
  ```
  Usługa nie dysponuje transakcją. Tryb przydatny jeśli w przypadku niepowodzenia wykonania się usługi nie będzie to miało wpływu na funkcjonowanie całej aplikacji. Wszystkie wyjątki powinny zostać obsłużone w ramach usługi ponieważ nieobsłużone spowodują przerwanie transakcji klienta.

Protokół dwufazowego zatwierdzania

Każda transakcja na początku , dopóki jej zasięg nie przekracza zakresu domeny oraz danej usługi jest transakcją o zasięgu lokalnym. Dopóki wykorzystuje ona pojedynczy zasób trwały do jej zarządzania wykorzystywany jest LTM (ang. *Lightweight Transaction Manager*) wykorzystujący lekki protokół (*ang. lightweight protocol*). Ma ona przypisany identyfikator który przechowywany jest we właściwości `TransactionInformation.LocalIdentifier` obiektu aktualnej transakcji `Transaction.Current`. LTM oferuje największą wydajność. W przypadku kiedy wykorzystywany będzie jeden transakcyjny zasób jądra KRM (ang. *Kernel Resource Manager*), może to być przykładowo transakcyjny system plików (TxF) bądź transakcyjny rejestr (TxR) nastąpi automatyczny awans do menedżera KTM (*ang. Kernel Transaction Manager*) on również wykorzystuje lekki protokół. W momencie kiedy transakcja próbuje zarejestrować drugi zasób zasób jądra, bądź jej granice zaczynają przekraczać domenę aplikacji bądź granicę usługi (propagacja transakcji klienta do usługi bądź jednej usługi do innej) następuje jej automatyczne awansowane do menedżera DTC (*ang. Distributed Transaction Manager*). Jej identyfikator możemy odczytać z właściwości `TransactionInformation.DistributedIdentifier` bieżącej transakcji. Wyboru i przydzielenia odpowiedniego protokołu automatycznie dokona środowisko .NET. Nie musimy pisać żadnego kodu.

Działanie protokołu dwufazowego zatwierdzania stosowanego przez DTC jest nomen omen podzielone na dwie fazy. W pierwszej fazie następuje głosowanie. Algorytm sprawdza czy wszystkie zasoby są w stanie zatwierdzić swoje zmiany gdyby takie żądanie otrzymały. W drugiej fazie, jeśli wszystkie zasoby odpowiedziały pozytywnie – następuje zatwierdzenie zmian. W innym przypadku, w razie przynajmniej jednego zastrzeżenia – wszystkie zmiany zostają odwołane i system wraca do stanu sprzed transakcji. W celu utworzenia zasobu obsługującego automatyczną rejestrację w transakcji oraz protokół dwufazowego zatwierdzania należy zaimplementować interfejs `IEnlistmentNotification`. Wymaga on oprogramowania następujących metod:

- `Prepare` – metoda wywoływana w pierwszej fazie, kiedy manager transakcji odpytuje managerów zasobów czy są zdolni do zatwierdzenia transakcji.
- `Commit` – metoda wywoływana w drugiej fazie w momencie zatwierdzania transakcji.
- `Rollback` – metoda wywoływana w drugiej fazie w momencie odwołania transakcji.
- `InDoubt` – metoda wywoływana w drugiej fazie kiedy transakcja budzi wątpliwości, najprawdopodobniej nie udało się jej w pełni zatwierdzić oraz w pełni odwołać co mogło się przyczynić do rozspójnienia systemu.

Ponadto w konstruktorze należy zarejestrować menedżera zasobu trwałego w transakcji, wykorzystując metodę statyczną `Transaction.Current.EnlistDurable()`. Przykładowa klasa reprezentująca menedżera zasobu takiego jak plik wygląda następująco:

```
...
private string path;
private string content;
public TransactionalFileCreator(string path, string content) {
      this.path = path;
      this.content = content;
      Transaction.Current.EnlistDurable(Guid.NewGuid(), this,
                                        EnlistmentOptions.None);
      }
public void Commit(Enlistment enlistment) {
      File.WriteAllText(path, content);
      enlistment.Done();
      }
public void InDoubt(Enlistment enlistment) {
      enlistment.Done();
      }
public void Prepare(PreparingEnlistment preparingEnlistment) {
      preparingEnlistment.Prepared();
      }
public void Rollback(Enlistment enlistment) {
      if(File.Exists(path))
            File.Delete(path);
      enlistment.Done();
      }
...
```

Jawne zarządzanie przepływem transakcji

Domyślnie, poprawne wykonanie metody oznacza automatyczne zatwierdzenie transakcji. Jeśli powodzenie transakcji zależy od czegoś innego niż brak błędów bądź wyjątków możemy posłużyć się trybem jawnego zatwierdzania transakcji. Wymaga on wykorzystania sekcji transportowej, co możemy uzyskać za pomocą atrybutu kontraktu: `[ServiceContract(SessionMode = SessionMode.Required)]`. Metody w których ręcznie będziemy zatwierdzać sekcję należy opatrzyć atrybutem: `[OperationBehavior(TransactionScopeRequired = true, TransactionAutoComplete = false)]`. Zatwierdzenie transakcji następuje po wywołaniu metody `OperationContext.Current.SetTransactionComplete()`. Próba wykonania jakiejkolwiek czynności transakcyjnej po zatwierdzeniu transakcji np. próba dostępu do właściwości `Transaction.Current` zakończy się wyjątkiem.

Jawne programowanie transakcji

Jeżeli chcemy aby klient nie będący usługą posiadał własną transakcję możemy skorzystać z klasy `TransactionScope`. Zatwierdzenie transakcji odbywa się poprzez wywołanie metody `Complete` inaczej zostanie ona przerwana.

```
try {
      using (TransactionScope scope = new TransactionScope()) {
            // kod klienta
            scope.Complete();
            }
} catch(TransactionAbortedException e) {
      Console.WriteLine(e.Message);
} catch {
      Console.WriteLine("Transakcja przerwana");
      }
```

W zależności od wartości obiektu `TransactionScopeOption` przekazanego do konstruktora klasy `TransactionScope`:

- istniejąca transakcja będzie kontynuowana, a w przypadku jej braku zostanie utworzona nowa. Zachowanie domyślne (`TransactionScopeOption.Required`),
- zawsze zostanie utworzona nowa transakcja (opcja: `TransactionScopeOption.RequiresNew`),
- bieżąca transakcja – jeśli istnieje – zostanie stłumiona (opcja: `TransactionScopeOption.Suppress`).

Dzięki dwóm ostatnim opcjom możliwość wykonania wewnątrz transakcji pewnych operacji w osobnej transakcji albo trybie nietransakcyjnym. Przykładowo wykonanie kodu nietransakcyjnego mogło by wyglądać następująco:

```
try {
      using(TransactionScope scope =
            new TransactionScope(TransactionScopeOption.Suppress)) {
            // tutaj operacje zakończone wyjątkiem
            // nie będą miały wpływu na zachowanie transakcji
            throw new Exception();
            }
} catch { ... }
```

Błędy

… service located at … is unavailable. This could be because the service is too busy or because no endpoint was found listening at the specified address. Please ensure that the address is correct and try accessing the service again later
może wystąpić podczas uruchamiania kilku usług jednocześnie (opcja – multiple startup projects:). Uruchom wymagane usługi a następnie klienta ręcznie klikając ppm na projekcie i wybierając opcję: Debug → Start new instance.

The service operation requires a transaction to be flowed
wymagana jest transakcja klienta.

The HttpGetEnabled property of ServiceMetadataBehavior is set to true and the HttpGetUrl property is a relative address, but there is no http base address. Either supply an http base address or set HttpGetUrl to an absolute address
jeśli metadane udostępniane są po innym protokole niż http a adres w usłudze mex ustawiony jest jako adres względny to należy wyłączyć udostępnianie metadanych za pomocą metody GET protokołu Http ustawiając wartość właściwości HttpGetEnabled na true.

The type name '...' does not exist in the type '...'
Nazwy przestrzeni nazw i klasy programu są takie same i kod wygenerowanego proxy nie potrafi ich rozróżnić. Należy zmienić nazwę klasy programu.