Windows Communication Foundation

Cuvinte cheie

Spatii de nume folosite in WCF
Arhitectura WCF
Adresa
Binding
Contract
Endpoint
Host
Client
Fisiere de configurare
Clasa ServiceHost
Exemple concrete de implementare WCF
- metadata prin interfata ;
- metadata cu svcutil.exe

Tipuri de operatii
- sincrone
- asincrone
- one-way
- duplex

Comunicare bidirectionala. Modalitati de implementare
Exemplu complet implementat
Fisiere de configurare

Atribute
- [ServiceContract]
- [OperationContract]

Bibliografie:

1. **Pablo Cibraro, Kurt Claeys, Fabio Cozzolino, Johan Grabner: Profesional WCF 4 Windows Communication Foundation with .NET 4**

2. **John Sharp: Windows Communication Foundation 4, Step by Step**

3. **Juval Lowy: Programming WCF Services**

4. **Steve Resnick, Richard Crane, Chris Bowen : Essential Windows Communication Foundation For .NET 3.5**

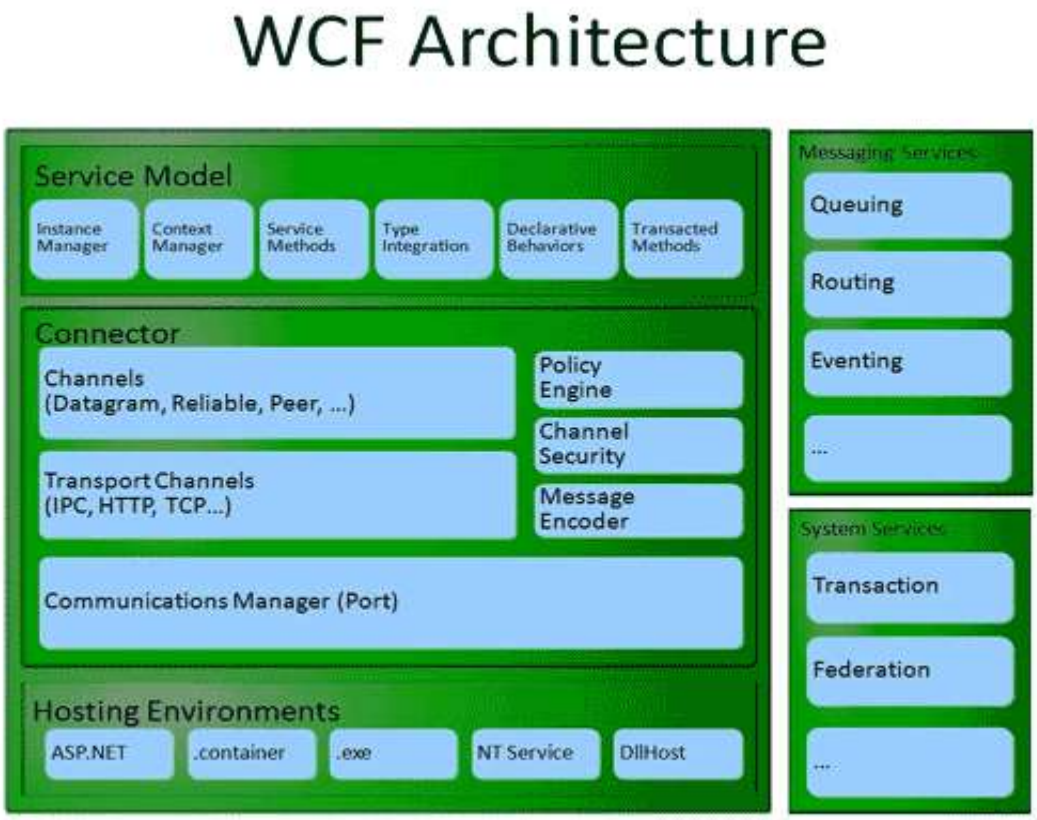
Modelul Windows Communication Foundation - WCF

Modelul de programare WCF unifica diversele modele de programare in care aplicatiile pot comunica intre ele. .NET furnizeaza API separat pentru comunicatii bazate pe SOAP in vederea unei interoperabilitati maxime (Web Services), comunicatii binare intre aplicatii ce ruleaza pe masini Windows (.NET Remoting), comunicatii tranzactionale (Distributed Transactions) si comunicatii asincrone (Message Queues). Toate aceste modele sunt inglobate intr-un singur model general, programare orientata pe servicii.

WCF este proiectat avand in vedere principiile arhitecturii orientate pe servicii pentru a suporta calculul distribuit ("*Distributed computing*") unde serviciile sunt consumate de clienti. Clientii pot consuma servicii multiple si un serviciu poate fi consumat de mai multi clienti. Serviciile au o interfata WSDL. WFC implementeaza mai multe servicii Web standard cum ar fi: WS-Addressing, WS-ReliableMessaging si WS-Security.

Modelele de programare ale aplicatiilor ce au nevoie sa comunice intre ele raman in continuare viabile. Putem programa utilizand WCF sau celelalte modele.

WCF consta dintr-o multime de clase noi: Service Model, Channel Layer, XML Formatter, Extensible Security Infrastructure. De asemenea s-au adaugat functionalitati pentru gazduirea solutiilor WCF in IIS (Internet Information Service), serverul Web ce este construit in cadrul SO Windows.



- Un serviciu reprezinta unitatea functionala expusa spre a fi utilizata.
- Un serviciu poate fi local sau remote.
- Clientii si serviciile interactioneaza prin trimiterea / primirea de *mesaje*.

Un serviciu poate fi vazut ca o multime de **endpoints**.

Un **endpoint** este o resursa pe retea, unde pot fi trimise mesaje.

Serviciile asculta pentru mesaje pe adresa specificata de **endpoint** si asteapta ca mesajul sa ajunga intr-un format particular (pe care stie sa-l interpreteze).

FIGURE 1.1 Communication between client and service

In WCF toate mesajele sunt SOAP.

Clientii nu interactioneaza cu serviciile in mod direct, se utilizeaza un **proxy** chiar daca sunt in acelasi **AppDomain** (acest lucru nu este valabil in .NET Remoting).

Nucleul de baza pentru WCF este furnizat de urmatorii assemblies:

- **System.Runtime.Serialization.dll** : defineste spatiile de nume si tipurile ce pot fi folosite pentru serializarea si deserializarea obiectelor in WCF.
- **System.ServiceModel.dll** : assembly ce contine tipurile folosite pentru a construi aplicatii WCF.

In cadrul acestor assemblies sunt definite o serie de noi spatii de nume dintre care enumeram (cele mai des folosite) :

System.Runtime.Serialization : contine clase ce pot fi utilizate pentru serializarea si deserializarea obiectelor. Serializarea este procesul de conversie al unui obiect intr-o secventa liniara de *bytes* folosita in continuare fie pentru memorare, fie pentru transmitere la o alta locatie. Deserializarea este procesul invers ce consta din recrearea obiectelor din sirul de *bytes* primiti.Interfata **ISerializable** furnizeaza o metoda pentru a controla comportarea serializarii instantelor tipurilor.

Clasele din spatiul de nume **System.Runtime.Serialization.Formatters** controleaza formatarea actuala a diferitelor tipuri de date incapsulate in obiectele serializate.

Clase pentru contractul de date

Clasele **DataContractAttribute** (atribut aplicat la clase) si **DataMemberAttribute** (atribut aplicat la date membru) sunt folosite in momentul cand cream servicii pentru a specifica ce membrii ai clasei trebuie serializati.

O alta clasa importanta este **DataContractSerializer**, folosita pentru a serializa sau deserializa instantele claselor ce au atributul **DataContractAttribute** si a membrilor acestora adnotati cu atributul **DataMemberAttribute**.

- **`System.ServiceModel`**: contine clase, enumerari si interfete necesare pentru a construi un serviciu si aplicatii client ce pot fi folosite intr-un mediu distribuit de aplicatii.

- **`System.ServiceModel.Configuration`**: citire fisiere de configurare.
- **`System.ServiceModel.Description:`** contine tipuri, enumerari si interfete necesare pentru a construi si modifica descrierile serviciilor, contracte, endpoint-uri, etc.
- **`System.ServiceModel.Security:`** controlul securitatii serviciilor in WCF.
- **`System.ServiceModel.MsmqIntegration:`** contine tipuri necesare pentru integrarea cu servicii MSMQ.

Schema generala a unei aplicatii ce foloseste WCF (clientul si serviciul pot fi sau nu pe aceasi masina) poate fi descrisa ca mai jos:

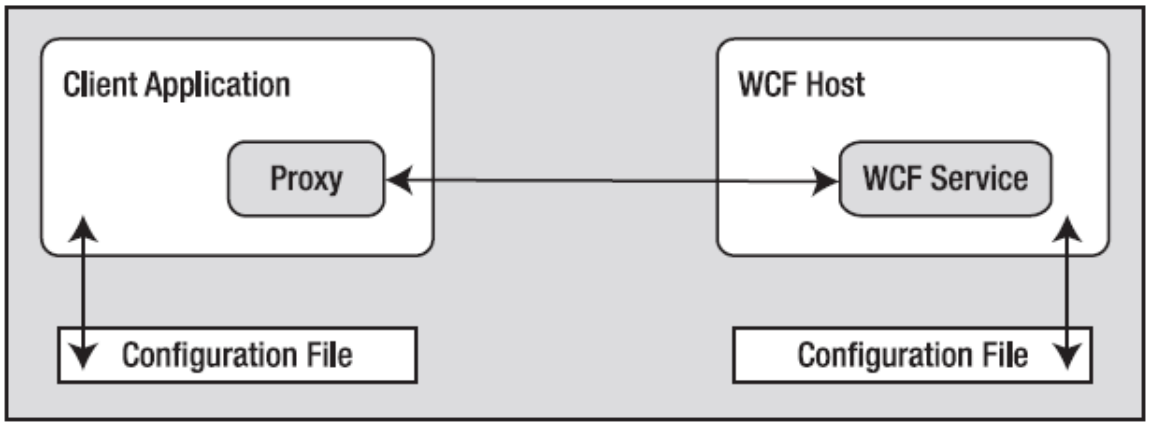


Figure 25-4. *A high-level look at a typical WCF application*

Partile componente ale unui serviciu

(A, B, C) = (Adresa, Binding, Contract)

A = Adresa (Unde?)

In WCF, fiecare serviciu este asociat cu o adresa. Aceasta adresa furnizeaza urmatoarele informatii:
- locatia unde se gaseste serviciul ;
- protocolul de transport sau schema de transport folosita pentru a comunica cu serviciul.

WCF suporta urmatoarele scheme de transport:
- HTTP
- TCP
- Peer network
- IPC (Inter-Process Communication peste pipe-uri cu nume)
- MSMQ

Adresele au urmatorul format:

```
[base address]/[optional URI]
```

unde "**base address**" este totdeauna in formatul:

```
[transport]://[machine or domain][:optional port]
```

Exemple:

```
http://localhost:8001/MyService
```

Urmatoarea schema

```
http://localhost:8001/MyService
```

poate fi interpretata astfel:

Mergi pe masina numita « localhost » unde pe portul 8001 cineva cu numele MyService asteapta apeluri de la mine.

Conform schemelor de transport putem distinge :

Schema transport	Schema URI	Observatie	Port implicit
TCP	net.tcp	Pot fi partajate. net.tcp://localhost :**1234**/MyService net.tcp://localhost :**1234**/MyNewService	808
HTTP	http sau https	Pot fi partajate. http://localhost:1234/MyService	80
IPC	net.pipe	Named pipe. Pe aceeasi masina. Nu pot fi partajate. net.ipc://localhost :1234/MyService	
MSMQ	net.msmq	Trebuie specificat numele cozii. **net.msmq://localhost/private/MyService** **net.msmq://localhost/MyService**	
Peer to Peer	net.p2p		

Observatie
URI-urile ce incep cu ***net***. sunt specifice pentru Microsoft. Nu sunt recunoscute de toate platformele.

B = Binding (Cum ?)

Binding-urile definesc mecanismul de comunicatie ce trebuie folosit cand comunicam cu un endpoint si modul de conexiune la un endpoint. Un binding contine urmatoarele elemente:

- Stiva protocol determina securitatea, increderea si setarile contextului folosit pentru mesajele ce sunt trimise la endpoint.
- Transportul determina protocolul de transport folosit cand trimitem mesaje la endpoint, de exemplu, TCP sau HTTP.
- Codificarea determina codificarea in retea folosita pentru mesaje ce sunt trimise la endpoint, de exemplu, text/XML, binar, MTOM (Message Transmission Optimization Mechanism).

Exista mai multe pattern-uri pentru comunicatie:

- Mesaje sincrone cerere/raspuns sau asincrone de tip « trimite si nu asteapta raspuns ».
- Mesaje bidirectionale.
- Mesaje livrate imediat sau puse in coada. Cozile pot fi durabile sau volatile.

Protocoale de transport sunt: HTTP (sau HTTPS), TCP, P2P, IPC sau MSMQ.
In ceea ce priveste codificarea mesajelor exista urmatoarele posibilitati:

- *Plain text* pentru a permite interoperabilitate.
- *Codificare binara* pentru a optimiza performanta.
- *MTOM* (*Message Transport Optimization Mechanism*) pentru mesaje mari.

Referitor la securitatea mesajelor, se disting urmatoarele situatii :

- nesecurizate ;
- securitate la nivel de transport ;
- mesaj privat si securizat.

Pentru clienti exista posibilitati pentru autentificare si autorizare. Mesajele trimise pot fi de incredere sau nu si pot fi procesate in ordinea in care au fost trimise sau in ordinea in care au fost receptionate. Toate aceste complexitati legate de comunicatie si optiunile existente fac constructia unui client si al unui serviciu sa fie neproductiva. Pentru a simplifica aceste alegeri, WCF grupeaza impreuna o multime de aspecte de comunicatie in ceea ce se numeste ***binding***.

In concluzie un ***binding*** contine o multime de alegeri in ceea ce priveste *protocolul* de transport, *codificarea* mesajelor, pattern-ul de comunicatie, increderea, securitatea, propagarea tranzactiilor si interoperabilitatea.

Se pot folosi ***binding***-uri preconstruite in WCF sau putem scrie unul nou.
Serviciul publica ***binding***-ul in metadata, ceea ce permite clientilor sa o citeasca si sa foloseasca aceleasi valori ca si serviciul.

Un singur serviciu poate suporta *binding*-uri multiple pe adrese separate.

Binding-uri standard definite in WCF

Binding de baza

Oferit de clasa **`BasicHttpBinding`**, proiectat pentru a expune un serviciu WCF ca un serviciu web ASMX, deci clientii vechi vor putea folosi acest serviciu.

Elementul binding: **`basicHttpBinding`**.

In exemplul ce urmeaza se exemplifica folosirea clasei BasicHttpBinding si a elementului basicHttpBinding in cadrul fisierului de configurare.

Exemplu cu folosirea clasei **BasicHttpBinding**

```
BasicHttpBinding binding = new BasicHttpBinding();
binding.Name = "binding1";
binding.HostNameComparisonMode = HostNameComparisonMode.StrongWildcard;
binding.Security.Mode = BasicHttpSecurityMode.None;

Uri baseAddress =
        new Uri("http://localhost:8000/servicemodelsamples/service");
Uri address =
        new Uri("http://localhost:8000/servicemodelsamples/service/calc");
```

```
// Creare ServiceHost pentru tipul CalculatorService
// Se furnizeaza adresa de baza.
ServiceHost serviceHost = new ServiceHost(typeof(CalculatorService),
        baseAddress);

serviceHost.AddServiceEndpoint(typeof(ICalculator), binding, address);

// Apel metoda Open din ServiceHostBase pentru a crea "listeners"
// si a incepe "ascultarea" pentru mesaje.
serviceHost.Open();

// In acest moment serviciul poate fi accesat.
Console.WriteLine("The service is ready.");
Console.WriteLine("Press <ENTER> to terminate service.");
Console.ReadLine();

// Stop (inchidere) serviciu.
serviceHost.Close();
```

Exemplu de folosire in fisiere de configurare. Are acelasi efect ca si codul de mai sus,

```
<system.serviceModel>
  <services>
    <service
        type="Wcf.CalculatorService"
        behaviorConfiguration="CalculatorServiceBehavior">
       <endpoint address=""
            binding="basicHttpBinding"
            bindingConfiguration="Binding1"
            contract="Wcf.ICalculator" />
    </service>
  </services>
  <bindings>
     <basicHttpBinding>
        <binding name="Binding1"
              hostNameComparisonMode="StrongWildcard">
              <security mode="None" />
         </binding>
     </basicHttpBinding>
  </bindings>
</system.serviceModel>
```

Binding TCP

Oferit de clasa **`NetTcpBinding`**. Foloseste comunicatia TCP intre masini pe intranet. Suporta incredere, tranzactii si securitate si este optimizat pentru comunicare WCF –to-WCF. Este necesar ca atat clientul cat si serviciul sa foloseasca WCF.

Elementul binding: **`netTcpBinding`**.

Binding Peer network

Oferit de clasa **`NetPeerTcpBinding`**.

Elementul binding: **`netPeerTcpBinding`**.

Binding IPC

Oferit de clasa **`NetNamedPipeBinding`**, folosit in comunicarea pe aceeasi masina. Este cel mai sigur.

Elementul binding: **`netNamedPipeBinding`**.

Binding Web Service (WS)

Oferit de clasa **`WSHttpBinding`**, foloseste HTTP sau HTTPS pentru transport, si este proiectat sa ofere o varitate de trasaturi: incredere, tranzactii si securitate in Internet.

Elementul binding : **`wsHttpBinding`**.

Binding Federated WS

Oferit de clasa **`WSFederationHttpBinding`**, este o dezvoltare pentru binding WS si ofera suport pentru securitate pe masini diferite folosind “ticket” sau “token”.

Elementul binding: **`wsFederationHttpBinding`**.

Binding Duplex WS

Clasa **`WSDualHttpBinding`**. Este similar cu binding WS dar in plus suporta comunicarea bidirectionala.

Elementul binding: **`wsDualHttpBinding`**.

Binding MSMQ

Clasa **`NetMsmqBinding`**. Foloseste MSMQ pentru transport si este proiectat sa ofere suport pentru servicii ce folosesc cozi de mesaje. Nu e necesar ca apelatul sa fie online.

Elementul binding: **`netMsmqBinding`**.

Binding MSMQ integration

Clasa **`MsmqIntegrationBinding`**. Proiectata pentru a lucra cu vechii clienti MSMQ.

Elementul binding : **`msmqIntegrationBinding`**.

Observatie

Incepand cu versiunea 8 a sistemului de operare Windows, s-a implementat binding **NetHttpBinding** si **NetHttpsBinding**. Aceste binding-uri lucreaza cu WebSocket, versiunea 2.

Elementul binding : **netHttpBinding**

NetHttpBinding este proiectat pentru a consuma servicii HTTP sau WebSocket si foloseste, in mod implicit, codificarea binara. NetHttpBinding va detecta daca este folosit cu un contract cerere-raspuns sau duplex si isi va schimba comportarea pentru a fi in concordanta cu tipul de contract. Se foloseste HTTP pentru cerere-raspuns si WebSockets pentru duplex. Aceasta comportare poate fi modificata prin setarea proprietatii WebSocketTransportUsage.

Formatare si Codificare (Encoding)

Fiecare din *binding*-urile standard foloseste metode de transport si codificari diferite.

Table 1-1. Transport and encoding for standard bindings (default encoding is in bold)

Name	Transport	Encoding	Interoperable
BasicHttpBinding	HTTP/HTTPS	**Text**, MTOM	Yes
NetTcpBinding	TCP	Binary	No
NetPeerTcpBinding	P2P	Binary	No
NetNamedPipeBinding	IPC	Binary	No
WSHttpBinding	HTTP/HTTPS	**Text,** MTOM	Yes
WSFederationHttpBinding	HTTP/HTTPS	**Text**, MTOM	Yes
WSDualHttpBinding	HTTP	**Text**, MTOM	Yes
NetMsmqBinding	MSMQ	Binary	No
MsmqIntegrationBinding	MSMQ	Binary	Yes

O codificare bazata pe text permite unui serviciu WCF (sau client) sa comunice peste HTTP cu orice alt serviciu (sau client) *indiferent de tehnologie*.

Codificarea binara peste TCP sau IPC are cea mai buna performanta dar reduce din interoperabilitate, trebuie sa avem WCF-la-WCF.

C = Contract (Ce ?)

In WCF, toate serviciile expun contracte. Contractul este folosit pentru a descrie ceea ce face serviciul. WCF defineste patru tipuri de contracte:

Contracte pentru servicii - `[ServiceContract]`

Descrie operatiile din cadrul serviciului pe care le poate executa clientul.

Contracte de date - [DataContract]

Defineste tipurile de data ce sunt transferate la si de la serviciu. WCF defineste in mod implicit contracte pentru tipurile preconstruite `int`, `string`, etc. dar se pot defini in mod explicit contracte de date pentru tipuri *custom*.

Un contract de date mapeaza tipurile CLR la XML Schema Definitions (XSD) si defineste cum sunt serializate si deserializate datele.

Contracte pentru erori - [FaultContract]

Defineste erorile semnalate de serviciu si modul cum serviciul manipuleaza si propaga erorile catre clienti.

Contracte pentru mesaj - [MessageContract]

Permite serviciului sa interactioneze in mod direct cu mesajele. Contractele pentru mesaj pot fi *cu tip* sa *fara tip*. Se mapeaza tipurile CLR la mesajele SOAP si se descrie formatul mesajului SOAP si modul cum afecteaza definitiile WSDL si XSD ale acestor mesaje. Contractul pentru mesaj– furnizeaza un control precis asupra header-ului si corpului mesajului SOAP.

Elemente din WSDL – Web Service Definition Language

Element WSDL	Descriere
Type	Definitiile tipurilor de date folosite pentru a descrie schimbul de mesaje. Acestea sunt sub forma unei scheme de definitie XML.
Message	Reprezinta o definitie abstracta a datelor ce vor fi transmise. Un mesaj consta din parti logice, fiecare parte fiind asociata cu o definitie din interiorul unui tip sistem. Un mesaj este similar unui parametru formal al unei functii si este folosit pentru a defini prototipul (signatura) operatiilor.
Operation	Un nume si o descriere a unei actiuni suportate de serviciu. Operatiile expun capabilitatea sau functionalitatea unui endpoint al serviciului.
PortType	Un endpoint din serviciu implementeaza un **`PortType`**, ce grupeaza operatii similare.
Binding	Defineste formatul mesajului si detaliile protocolului pentru operatiile si mesajele definite de un **`PortType`** particular.
Port	Defineste un endpoint individual prin specificarea unei singure adrese pentru un binding.
Service	Defineste o multime de porturi relationate.

Trebuie sa existe o corespondenta intre contract si codul din CLR.

Contractul este descris in **WSDL** si **XSD,** iar codul lucreaza cu tipuri **CLR**. WCF faciliteaza aceasta mapare astfel :

1. Cand scriem codul in serviciu vom decora clasele cu atributele **[ServiceContract]**, **[OperationContract]**, **[FaultContract]**, **[MessageContract]** si/sau **[DataContract]**.
2. Cand scriem codul in client, cerem serviciului detalii despre contracte si sa ne genereze o clasa *proxy* ce expune interfata serviciului, pe care o putem apela din cod. Acest lucru poate fi facut din Visual Studio sau folosind utilitarul **svcutil.exe**.
3. La runtime cand clientul apeleaza o metoda pe interfata serviciului, WCF serializeaza tipurile CLR si apelurile de metoda in XML si trimite mesajul in concordanta cu binding-ul, schema de codare agreata de WSDL.

Utilitare necesare:

- **svcutil.exe** – ce poate fi apelat din linia de comanda ;
- ***Add Service Reference*** din Visual Studio**.**

Acesta produce WSDL si genereaza clasele proxy ce faciliteaza maparea dintre tipurile .NET si XSD, si dintre metodele din clasa .NET si operatiile WSDL.

- **SvcTraceViewer.exe**, Service Trace Viewer, utilitar grafic ce citeste si interpreteaza diagnosticile din fisierul de log-uri scris de WCF. Putem vedea astfel formatul mesajelor primite si trimise de endpoints si ordinea acestora.

Contractul pentru serviciu – [ServiceContractAttribute]

Un contract pentru serviciu descrie interfata catre operatiile implementate de un serviciu endpoint.
O clasa adnotata cu [ServiceContract] si metodele sale cu **[OperationContract]** este expusa in WSDL ca **wsdl:service** si **wsdl:operation**.

Exemplu :

```
[ServiceContract]
public class StockService
{
   [OperationContract]
   double GetPrice(string ticker)
   {
         return 94.85;
   }
}
```

in WSDL arata astfel:

```
<wsdl:definitions ... >
      <wsdl:types> ... </wsdl:types>
      <wsdl:message
            name="StockService_GetPrice_InputMessage">
            <wsdl:part .. element="tns:GetPrice" />
      </wsdl:message>
      <wsdl:message
            name="StockService_GetPrice_OutputMessage">
            <wsdl:part .. element="tns:GetPriceResponse" />
```

```
        </wsdl:message>
        <wsdl:portType name="StockService">
              <wsdl:operation name="GetPrice"> ..
              </wsdl:operation>
        </wsdl:portType>
        <wsdl:service name="StockService">
              <wsdl:port name="BasicHttpBinding_StockService"
                    <soap:address
                    location="http://localhost/RequestResponse/
                    StockService.svc" />
              </wsdl:port>
        </wsdl:service>
  </wsdl:definitions>
```

Exemplu. Implementare contract pentru serviciu. Se folosesc interfete.

```
[ServiceContract]
interface IMyContract
{
   // metoda este parte a contractului
   [OperationContract]
   string MyMethod(string text);

   //Metoda nu este parte a contractului
   string MyOtherMethod(string text);
}

// clasa ce implementeaza interfata
class MyService : IMyContract
{
   public string MyMethod(string text)
   {
      return "Hello " + text;
   }
   public string MyOtherMethod(string text)
   {
      return "Nu pot apela aceasta metoda in WCF";
   }
}
```

Atributul **`ServiceContract`** se aplica pe ***interfete*** sau ***clase*** si face ca tipul respectiv sa fie un contract WCF.

Vizibilitatea este un concept al CLR si nu al WCF.

Aplicand acest atribut pe **`internal interface`** face ca aceasta interfata sa fie ca un contract « public » al serviciului. Fara atributul [**`ServiceContract`**] interfata nu este vizibila clientilor WCF.

Pentru a evita coliziunile de nume putem folosi *namespace* ca mai jos:

```
[ServiceContract(Namespace = "MyNamespace")]
```

si mai poate fi folosita proprietatea ***Name*** din [**OperationContract**].

In interiorul clasei trebuie sa definim metodele ce sunt vizibile (ce pot fi folosite) clientilor, folosind atributul [**OperationContract**] definit astfel:

```
[AttributeUsage(AttributeTargets.Method)]
public sealed class OperationContractAttribute : Attribute
{
   public string Name // evitare coliziune de nume si nu numai
   {get;set;}
   // alti membri
}
```

[**OperationContract**] se aplica numai metodelor.

O metoda decorata cu [**OperationContract**] nu poate folosi ca parametri referinte la obiecte.

Se admit numai tipuri primitive sau "*contracte de date*".

[**OperationContract**] se poate aplica atat pe metode publice cat si private.

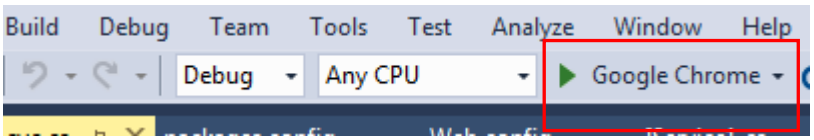
Exemplu definire contract de date – cod generat de VS cand se creaza un serviciu WCF.
Am modificat numele interfetei si implementarea metodei *GetDataUsingDataContract*.

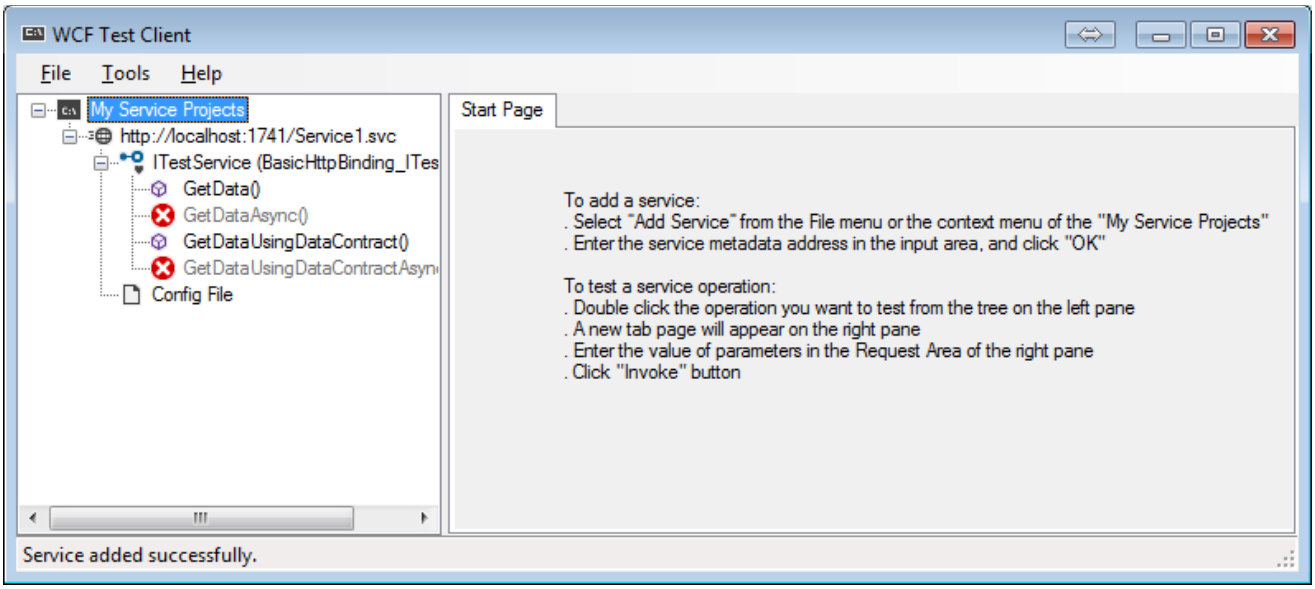
```
[ServiceContract]
public interface ITestService
{

    [OperationContract]
    string GetData(int value);

    [OperationContract]
    CompositeType GetDataUsingDataContract(CompositeType composite);
}


// Use a data contract as illustrated in the sample below to add composite types
// to service operations.
[DataContract]
public class CompositeType
{
    bool boolValue = true;
    string stringValue = "Hello ";

    [DataMember]
    public bool BoolValue
    {
        get { return boolValue; }
        set { boolValue = value; }
    }

    [DataMember]
    public string StringValue
    {
        get { return stringValue; }
        set { stringValue = value; }
    }    }
```

Interfata **ITestService** este implementata in continuare:

```
public class ServiceTest : ITestService
    {
        public string GetData(int value)
        {
            return string.Format("You entered: {0}", value);
        }

        public CompositeType GetDataUsingDataContract(CompositeType composite)
        {
            if (composite == null)
            {
                //throw new ArgumentNullException("composite");
                composite = new CompositeType();
                composite.StringValue = "Instanta creata de serviciu WCF ...";
            }
            if (composite.BoolValue)
            {
                composite.StringValue += "Suffix";
            }
            return composite;
        }
```

Deoarece am ales un proiect de tip WCF Service, acest serviciu va fi gazduit de IIS. Cand lansam in executie acest serviciu, din browser se lanseaza utilitarul WcfTestClient ca in figura urmatoare :



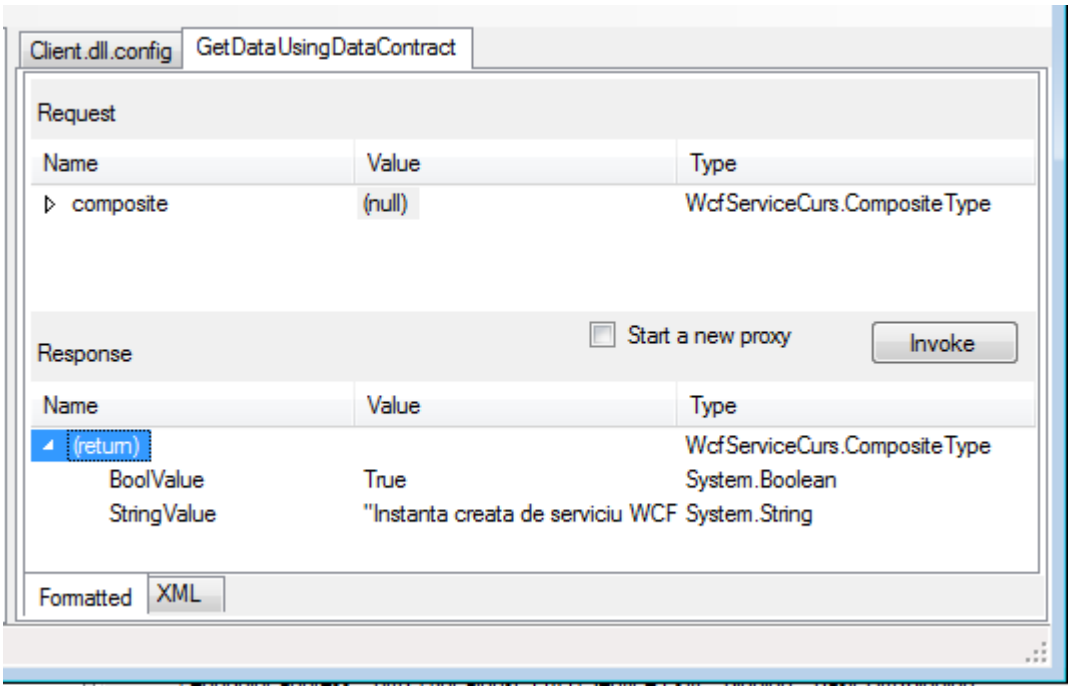


Observam in fereastra My Service Projects adresa ce poate fi folosita in browser : http://localhost:1741/Service1.svc, serviciul ITestService ce expune metodele GetData() si GetDataUsingDataContract() si fisierul de configurare Config File.
Daca selectam Config File obtinem urmatoarele informatii :

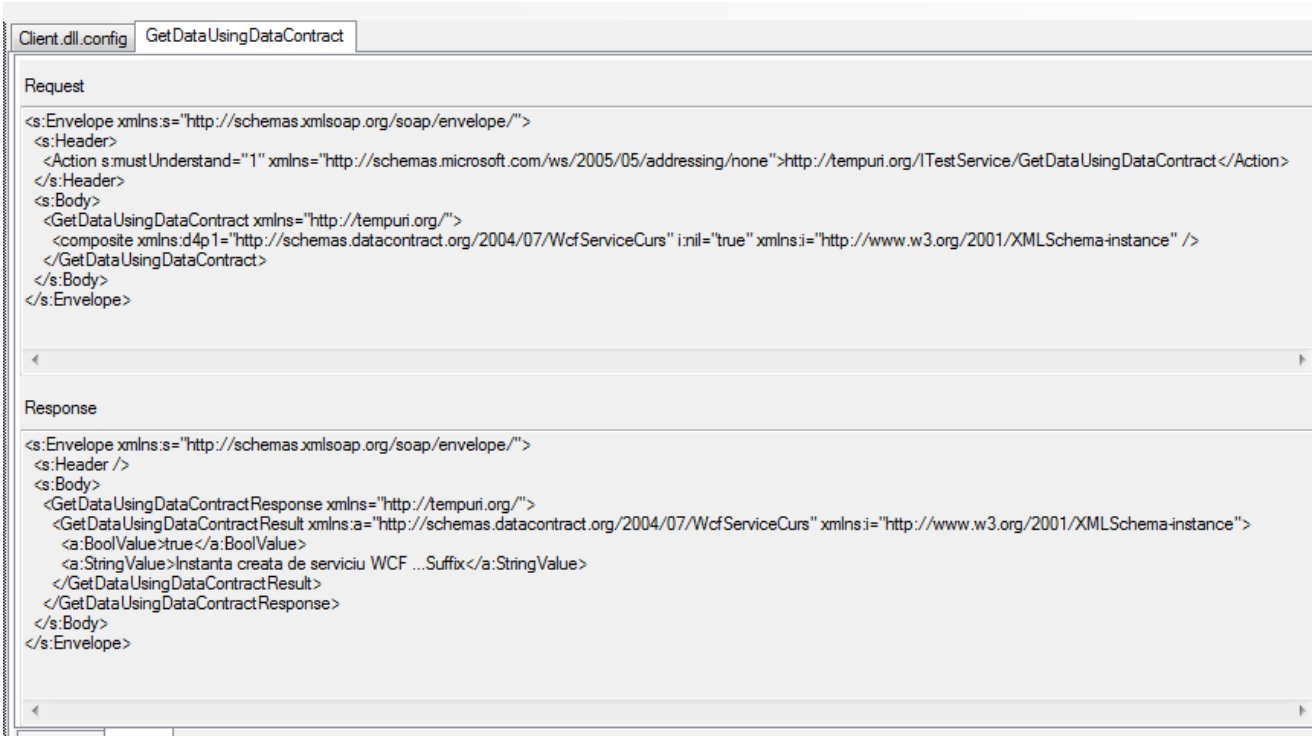
Client.dll.config

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.serviceModel>
    <bindings>
      <basicHttpBinding>
        <binding name="BasicHttpBinding_ITestService" sendTimeout="00:05:00" />
      </basicHttpBinding>
    </bindings>
    <client>
      <endpoint address="http://localhost:1741/Service1.svc" binding="basicHttpBinding"
        bindingConfiguration="BasicHttpBinding_ITestService" contract="ITestService"
        name="BasicHttpBinding_ITestService" />
    </client>
  </system.serviceModel>
</configuration>
```

Asa ar trebui sa arate fisierul de configurare pe partea clientului.
Apelam metoda GetDataUsingDataContract, selectam « null » pentru valoarea parametrului formal si obtine urmatorul rezultat.



si daca selectam modul de afisare XML obtinem :



Din figura se observa mesajul SOAP pentru cerere si pentru raspuns.

Hosting - Gazduire serviciu

Un serviciu WCF trebuie sa ruleze intr-un proces Windows. Un singur proces poate gazdui mai multe servicii si acelasi serviciu poate fi gazduit in mai multe procese.

Gazda serviciului poate fi furnizata de **IIS**, de **servicii Windows** sau de **aplicatii proprii** (ce pot fi CUI sau GUI).

IIS Hosting

Avantajul principal al unui asemenea *host* este acela ca procesul este lansat automat la prima cerere a clientului iar ciclul de viata al procesului *host* este gestionat de IIS.

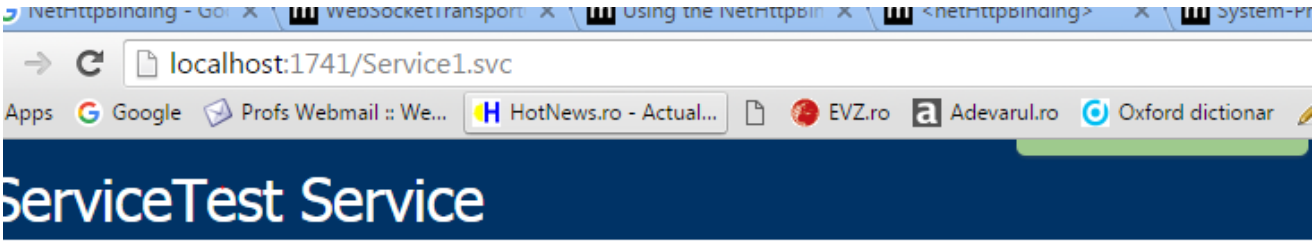
Dezavantaj : nu putem folosi decat HTTP.

Observatie. In **IIS5** trebuie folosit acelasi port de catre toate serviciile.

Pentru a gazdui un serviciu WCF in IIS trebuie sa cream un director virtual sub IIS si sa furnizam un fisier cu extensia **.svc** in care vom indica numele serviciului si clasa ce contine codul. Pentru exemplul prezentat mai sus acest fiser are urmatorul continut :

```
<%@ ServiceHost Language="C#" Debug="true"
Service="WcfServiceCurs.ServiceTest" CodeBehind="Service1.svc.cs" %>
```

Pentru exemplul anterior daca completam in browser adresa serviciului obtinem:



ou have created a service.

o test this service, you will need to create a client and use it to call the service. You can do this using th

```
svcutil.exe http://localhost:1741/Service1.svc?wsdl
```

You can also access the service description as a single file:

```
http://localhost:1741/Service1.svc?singleWsdl
```

his will generate a configuration file and a code file that contains the client class. Add the two files to yc
xample:

:#

```
class Test
{
    static void Main()
    {
        TestServiceClient client = new TestServiceClient();

        // Use the 'client' variable to call operations on the service.

        // Always close the client.
        client.Close();
    }
}
```

Fisierul Web.Config

Fisierul de configurare al site-ului (**web.config**) trebuie sa contina tipurile expuse ca servicii si va trebui sa calificam complet aceste tipuri.

```
<system.serviceModel>
   <services>
      <service name = "MyNamespace.MyService">
         ...
      </service>
   </services>
</system.serviceModel>
```

Self-Hosting

Self-hosting constituie tehnica folosita de dezvoltator pentru crearea si gestionarea procesului ce expune servicii WCF. Procesul poate fi pe aceeasi masina sau pe o masina din retea iar aplicatia poate fi serviciu Windows, CUI sau GUI. Serviciul poate fi gazduit si "in-proc", adica in acelasi proces cu clientul.

Procesul (serverul) trebui sa ruleze inaintea rularii clientului.

Fisierul de configurare al aplicatiei (**app.config**) trebuie sa listeze tipurile serviciilor pe care le gazduieste si le expune catre exterior.

```
<system.serviceModel>
   <services>
      <service name = "MyNamespace.MyService">
         ...
      </service>
   </services>
</system.serviceModel>
```

Crearea "host-ului" se face folosind clasa **ServiceHost**.

Exemplu.

Definire serviciu:

```
[ServiceContract]
interface ITestService
{...}
class ServiceTest : ITestService
{...}
```

iar pentru "host"(este o aplicatie CUI in acest caz):

```
public static void Main( )
{
   Uri baseAddress = new Uri("http://localhost:8000/");
   ServiceHost host = new ServiceHost(typeof(ServiceTest),baseAddress);

   host.Open( );

   // cod mentinere aplicatie in "viata"

   host.Close( );
}
```

Configurare *Endpoint*-uri

Un serviciu este asociat cu:

- **A** : adresa – **unde** e serviciul ;
- **B** : un *binding* – **cum** comunicam cu serviciul;
- **C** : un contract – **ce** face serviciul.

ABC-ul serviciului formeaza endpoint-ul.

In mod logic endpoint-ul este interfata serviciului care este analoaga cu interfata CLR sau COM.

Fiecare serviciu trebuie sa expuna *cel putin* un endpoint si fiecare endpoint are *exac*t un contract.

Toate endpoint-urile pe un serviciu au o adresa unica, si un singur serviciu poate expune endpoint-uri multiple. Aceste endpoint-uri pot folosi acelasi binding sau nu si pot expune acelasi contracte sau nu. Nu exista nici o relatie intre endpoint-uri si furnizorul de servicii.

Endpoint-urile sunt externe codului din serviciu. Endpoint-urile pot fi configurate folosind fisiere de configurare sau in mod programabil.

Configurare endpoint *folosind fisiere de configurare*

Exemplu

Definire serviciu

```
namespace WcfServiceCurs
{
    [ServiceContract]
    interface ITestService
    {...}
    class ServiceTest : ITestService
    {...}
}
```

Fisierul de configurare va arata astfel:

```
<system.serviceModel>
   <services>
      <service name = "WcfServiceCurs.ServiceTest">
         <endpoint
            address  = "http://localhost:8000/MyService/"
            binding  = "wsHttpBinding"
            contract = "WcfServiceCurs.ITestService"
         />
      </service>
   </services>
</system.serviceModel>
```

Observatie
Contractul si serviciul trebuie calificati complet.
Putem folosi aceeasi adresa de baza cu conditia ca URI sa fie diferit.

Exemplu cu endpoit-uri multiple pe acelasi serviciu

```
<system.serviceModel>
```

```
    <services>
      <service name = "WcfServiceCurs.ServiceTest">
         <endpoint
            address  = "http://localhost:8000/MyService/"
            binding  = "wsHttpBinding"
            contract = "WcfServiceTest.ITestService"
         />
         <endpoint
            address  = "net.tcp://localhost:8001/MyService/"
            binding  = "netTcpBinding"
            contract = " WcfServiceTest.ITestService "
          />
          <endpoint
            address  = "net.tcp://localhost:8002/MyService/"
            binding  = "netTcpBinding"
            contract = " WcfServiceTest.ITestService "
          />
      </service>
    </services>
</system.serviceModel>
```

Aceeasi adresa de baza dar URI diferit.

```
<system.serviceModel>
    <services>
      <service name = "WcfServiceCurs.ServiceTest">
         <endpoint
            address  = "net.tcp://localhost:8001/MyService/"
            binding  = "netTcpBinding"
            contract = " WcfServiceTest.ITestService "
         />
         <endpoint
            address  = "net.tcp://localhost:8001/MyOtherService/"
            binding  = "netTcpBinding"
            contract = " WcfServiceTest.ITestService"
         />
      </service>

      </services>
</system.serviceModel>
```

Din nou despre binding

Folosirea unui binding

Exista urmatoarele posibilitati de folosire a unui binding:

- unul preconstruit ;
- reconfigurarea unui *binding* preconstruit ;
- construirea unui *binding* nou.

Configurare binding

In fisierul de configurare vom folosi atributul **bindingConfiguration** in elementul <**endpoint** /> urmat de constructia sectiunii <**bindings** />. Sectiunea <**bindings** /> este element la nivel de <**system.serviceModel** />

Exemplu cu bindingConfiguration

In exemplul de mai jos se permite propagarea tranzactiilor.

```
<system.serviceModel>
   <services>
      <service name = "MyService">
         <endpoint
            address  = "net.tcp://localhost:8000/MyService/"
            bindingConfiguration = "TransactionalTCP"
            binding  = "netTcpBinding"
            contract = "ITestService"
         />
         <endpoint
            address  = "net.tcp://localhost:8001/MyService/"
            bindingConfiguration = "TransactionalTCP"
            binding  = "netTcpBinding"
            contract = "IOtherContract"
         />
      </service>
   </services>
   <bindings>
      <netTcpBinding>
         <binding name = "TransactionalTCP"
            transactionFlow = "true" />
      </netTcpBinding>
   </bindings>
</system.serviceModel>
```

Configurare binding programabil

A configura programabil un binding inseamna a folosi clasa corespunzatoare pentru binding si a adauga un endpoint cu acest binding.

Exemplu

```
ServiceHost host = new ServiceHost(typeof(ServiceTest));

NetTcpBinding tcpBinding = new NetTcpBinding();
tcpBinding.TransactionFlow = true;

host.AddServiceEndpoint(typeof(ITestService),tcpBinding,
                        "net.tcp://localhost:8000/ServiceTest");
host.Open( );
```

Configurare programabila Endpoint

Metoda folosita este **AddServiceEndpoint** din clasa **ServiceHost** ce are urmatorul prototip :

```
public class ServiceHost : ServiceHostBase
{
   public ServiceEndpoint AddServiceEndpoint(Type implementedContract,
                                             Binding binding,
                                             string address);
   //Additional members
}
```

Exemplu

```
ServiceHost host = new ServiceHost(typeof(ServiceTest));

Binding wsBinding  = new WSHttpBinding();
Binding tcpBinding = new NetTcpBinding();

host.AddServiceEndpoint(typeof(ITestService),
                        wsBinding,
                        "http://localhost:8000/MyService");
host.AddServiceEndpoint(typeof(ITestService),
                        tcpBinding,
                        "net.tcp://localhost:8001/MyService");
host.AddServiceEndpoint(typeof(IOtherContract),
                        tcpBinding,
                        "net.tcp://localhost:8002/MyService");
host.Open( );
```

Daca *host* furnizeaza o adresa de baza putem folosi si adrese relative.

```
// definire adresa de baza
Uri tcpBaseAddress = new Uri("net.tcp://localhost:8000/");

// folosire adresa de baza la creare instanta ServiceHost
ServiceHost host = new ServiceHost(typeof(ServiceTest),tcpBaseAddress);

Binding tcpBinding = new NetTcpBinding();

// Utilizare adresa de baza
host.AddServiceEndpoint(typeof(ITestService),tcpBinding,"");

// Adauga adresa relativa
host.AddServiceEndpoint(typeof(ITestService),tcpBinding,"MyService");

// Ignora adresa de baza
host.AddServiceEndpoint(typeof(ITestService),tcpBinding,
                        "net.tcp://localhost:8001/MyService");
host.Open( );
```

Metadata Exchange

Un serviciu are doua optiuni pentru a-si publica metadata :

- Utilizare protocol HTTP-GET ;
- Utilizare *endpoint* dedicat.

Folosind fisiere de configurare adaugam atributul **behaviorConfiguration** in sectiunea **service**, iar numele dat aici (atributul **name** din fisierul de configurare) pentru **behaviorConfiguration** il folosim in sectiunea <**behaviors**> unde completam elementul **<serviceMetadata />** ca in exemplul de mai jos:

```
<behaviors>
      <serviceBehaviors>
         <behavior name = "nume_dat_in_behaviorConfiguration">
            <serviceMetadata httpGetEnabled = "true"/>
         </behavior>
      </serviceBehaviors>
</behaviors>
```

iar fisierul de configurare complet arata astfel:

```
<system.serviceModel>
   <services>
      <service name = "MyService" behaviorConfiguration = "MEXGET">
         <host>
            <baseAddresses>
               <add baseAddress = "http://localhost:8000/"/>
            </baseAddresses>
         </host>
         ...
      </service>
      <service name = "MyOtherService"
                  behaviorConfiguration = "MEXGET">
         <host>
            <baseAddresses>
               <add baseAddress = "http://localhost:8001/"/>
            </baseAddresses>
         </host>
         ...
      </service>

</services>
   <behaviors>
      <serviceBehaviors>
         <behavior name = "MEXGET">
            <serviceMetadata httpGetEnabled = "true"/>
         </behavior>
      </serviceBehaviors>
   </behaviors>
```

```
</system.serviceModel>
```

Observatie
Daca schimbul de metadata este activat peste HTTP-GET putem folosi in browser adresa serviciului, de ex. http://localhost/MyService/Service1.svc si ne dam seama daca serviciul este instalat corect.

Endpoint pentru Metadata Exchange

Serviciul poate sa-si publice metadata printr-un endpoint special numit **MEX** (**Metadata EXchange endpoint**).

Acest endpoint suporta un standard pentru schimbul de metadata, reprezentat in WCF de interfata **`IMetadataExchange`**:

```
[ServiceContract(...)]
public interface IMetadataExchange
{
    [OperationContract(...)]
    Message Get(Message request);
    //More members
}
```

WCF implementeaza aceasta interfata si expune endpoint-ul pentru schimbul de metadata.

Tot ce trebuie sa facem este sa furnizam adresa, binding-ul de folosit si endpoint-ul pentru MEX.

Pentru binding-uri, WCF furnizeaza elemente de transport dedicate pentru HTTP, HTTPS, TCP si IPC. Adresa o putem da completa sau numai adresa de baza. Nu e nevoie sa activam optiunea HTTP-GET.

Exemplu

```
<service name = "MyService" behaviorConfiguration = "MEX">
   <host>
      <baseAddresses>
         <add baseAddress = "net.tcp://localhost:8001/"/>
         <add baseAddress = "net.pipe://localhost/"/>
      </baseAddresses>
   </host>
   <endpoint
      address  = "MEX"
      binding  = "mexTcpBinding"
      contract = "IMetadataExchange"
   />
   <endpoint
      address  = "MEX"
      binding  = "mexNamedPipeBinding"
      contract = "IMetadataExchange"
   />
   <endpoint
      address  = "http://localhost:8000/MEX"
      binding  = "mexHttpBinding"
      contract = "IMetadataExchange"
   />
</service>
<behaviors>
```

```
   <serviceBehaviors>
      <behavior name = "MEX">
         <serviceMetadata/>
      </behavior>
   </serviceBehaviors>
</behaviors>
```

Adaugare endpoint MEX in mod programabil

Trebuie sa construim un binding si apoi sa-l adaugam folosind metoda **AddServiceEndpoint**.
Numele contractului este **IMetadataExchange**.

```
// cod partial
BindingElement bindingElement = new TcpTransportBindingElement();
CustomBinding binding = new CustomBinding(bindingElement);

Uri tcpBaseAddress = new Uri("net.tcp://localhost:9000/");
ServiceHost host = new ServiceHost(typeof(ServiceTest),tcpBaseAddress);

ServiceMetadataBehavior metadataBehavior;
metadataBehavior =
    host.Description.Behaviors.Find<ServiceMetadataBehavior>();

// verificam sa nu fie deja in colectie
if(metadataBehavior == null)
{
   metadataBehavior = new ServiceMetadataBehavior( );
   host.Description.Behaviors.Add(metadataBehavior);
}
host.AddServiceEndpoint(typeof(IMetadataExchange),binding,"MEX");
host.Open( );
```

Clasa ServiceHost

Namespace: System.ServiceModel
Assembly: System.ServiceModel (in System.ServiceModel.dll)

Sintaxa

```
public class ServiceHost : ServiceHostBase{}
```

Observatie
Implementeaza host-ul folosit de modelul de programare al serviciului WCF.

Clasa **ServiceHost** se foloseste pentru a configura si expune un serviciu ce va fi utilizat de o aplicatie client in cazul cand nu folosim IIS (Internet Information Services) sau Windows Activation Services (WAS).

IIS si **WAS** interactioneaza cu obiectul **ServiceHost**.

Pentru a expune un serviciu, WCF are nevoie de o descriere completa a serviciului – reprezentata de clasa **`ServiceDescription`**.

Clasa **`ServiceHost`** creaza un obiect de tip **`ServiceDescription`** din tipul serviciului si informatiile de configurare si apoi foloseste aceasta descriere (obiect **`ServiceDescription`**) pentru a crea obiecte **`ChannelDispatcher`** pentru fiecare *endpoint* din descriere.

Un obiect **`ServiceHost`** este folosit pentru :

- Incarcare serviciu.
- Configurare *endpoints*.
- Aplicare setari de securitate.
- Lansare "*listner*" pentru a manipula cererile.

Constructori

Name	Description
`ServiceHost()`	Initializes a new instance of the ServiceHost class.
`ServiceHost(Object, Uri[])`	Initializes a new instance of the ServiceHost class *with the instance of the service* and its base addresses specified.
`ServiceHost(Type, Uri[])`	Initializes a new instance of the ServiceHost class *with the type of service* and its base addresses specified.

Descriere constructori

```
public ServiceHost(Object singletonInstance, params Uri[] baseAddresses)
```

unde :

`singletonInstance:` Instanta serviciului gazduit.

Instanta unui tip (un obiect) ce va fi utilizata de un serviciu singleton. In acest caz un apel al metodei **`InstanceContext.ReleaseServiceInstance`** nu are nici un efect cand o instanta a obiectului *well-known* este furnizata in constructor. Similar alte mecanisme de eliberare a instantei vor fi ignorate.

`ServiceHost` se comporta ca si cum proprietatea

`OperationBehaviorAttribute.ReleaseInstanceMode`

este setata cu valoarea **`ReleaseInstanceMode.None`** pentru toate operatiile.

`baseAddresses:` Un array de tip **Uri** ce contine adresele de baza pentru serviciul gazduit.

Observatie

Constructorul **`ServiceHost(Object, Uri[])`** se foloseste cand se doreste ca gazda serviciului sa foloseasca o instanta singleton specifica a serviciului.

Exemplu. Presupunem ca host-ul pentru serviciu este o aplicatie GUI si dorim ca fiecare metoda din cadrul serviciului, atunci cand este apelata, sa completeze anumite controale pe interfata serverului. In acest caz, din cadrul metodelor trebuie sa avem acces la fereastra aplicatiei.

O posibilitate de rezolvare este de a crea instanta singleton pentru serviciu, iar clasa ce implementeaza serviciul sa mentina un pointer la interfata pentru a apela metode din interfata.

```
// Am folosit aceasta varianta pentru a avea acces la interfata din cadrul
// metodellor definite in serviciu.
// Acest lucru e numai pentru exemplificare si vizualizare apel metode pe partea de server
// In mod normal serverul foloseste jurnalizare proprie (fisier de log-uri) unde se scriu
// informatii despre comportarea serviciului (eventuale erori).
ServicePost servicePost = new ServicePost();
host = new ServiceHost(servicePost,
                    new Uri("http://localhost:8080/httpPost"),
                    new Uri("net.tcp://localhost:8081/tcpBlog"));
try
{
     host.Open();

   // Am nevoie de obiectul curent (fereastra) this pentru a-l pasa in obiectul
   // de pe server. Fiecare metoda de pe server va afisa un mesaj in
   // ListBox in momentul cand va fi apelata.
   // Management instanta serviciu (atribut plasat pe clasa ce
   // implementeaza ServiceContract
   // [ServiceBehavior(InstanceContextMode = InstanceContextMode.Single)]
   singletonInstance = (IPost)host.SingletonInstance;
   ServicePost svp = (ServicePost)singletonInstance;
   svp.PassMainWindow(this);

// ...
} catch...
```

iar in clasa *ServicePost* – implementeaza interfata IPost - avem :

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.Single)]
public class ServicePost : IPost
{
        MainWindow mHwnd;

        public ServicePost()
        { }
        public void PassMainWindow(MainWindow mw)
        {
            mHwnd = mw;
        }

          public Post GetPostByTitle(string title)
        {
            mHwnd.WriteMessage("Apel GetPostByTitle: Valoare parametru = " + title);

            using (var context = new ModelPostBlogContainer())
            {
                context.Configuration.ProxyCreationEnabled = false;
                var post = context.Posts.Include(p => p.Comments)
                                .Single(p => p.Title == title);
                return post;
            }
        }
// cod
}
```

Metoda *WriteMessage*(*string*) definita pe server (actualizeaza un control ListBox) :

```
public  void WriteMessage(string msg)
{
     lbInfo.Items.Add(msg);
}
```

Alt prototip pentru ServiceHost :

```
public ServiceHost(Type serviceType, params Uri[] baseAddresses)
```

unde
serviceType: tipul serviciului gazduit.

Acest constructor se foloseste cand avem tipul serviciului si putem crea oricand noi instante ale tipului, chiar si atunci cand dorim o instanta singleton.

Exemplu

```
ServiceHost host = new ServiceHost(typeof(ServicePost), baseAddress);
```

Proprietati

Nume	Descriere
Authentication	Gets the service authentication behavior.
Authorization	Gets the authorization behavior for the service hosted.
BaseAddresses	Gets the base addresses used by the hosted service.
ChannelDispatchers	Gets the collection of channel dispatchers used by the service host.
Credentials	Gets the credential for the service hosted.
Description	Gets the description of the service hosted.
ImplementedContracts	Retrieves the contracts implemented by the service hosted.
IsDisposed	Gets a value that indicates whether the **communication object** has been disposed
SingletonInstance	Gets the singleton instance of the hosted service.
State	Gets a value that indicates the current state of the communication object. (Inherited from CommunicationObject.)

Metode

Name	Description
`AddBaseAddress()`	Adds a base address to the service host.
`AddDefaultEndpoints()`	Adds service endpoints for all base addresses in each contract found in the service host with the default binding.
`AddServiceEndpoint(ServiceEndpoint)`	Adds the specified service endpoint to the hosted service.
`AddServiceEndpoint(String, Binding, String) ...`	Adds a service endpoint to the hosted service with a specified contract, binding, and endpoint address.
`Close(); Close(TimeSpan)`	Causes a communication object to transition from its current state into the closed state. Causes a communication object to transition from its current state into the closed state within a specified interval of time. (Inherited from CommunicationObject.)
`Open(); Open(TimeSpan);`	Causes a communication object to transition from the created state into the opened state.(Inherited from CommunicationObject.)
`SetEndpointAddress()`	Sets the endpoint address of the specified endpoint to the specified address. (Inherited fromServiceHostBase.)

Observatie

- Exista si metode asincrone pentru **Open()** si **Close()** asupra serviciului.
- De remarcat multitudinea prototipurilor pentru adaugarea de endpoint-uri : vezi metoda **AddServiceEndpoint** precum si metoda **AddDefaultEndpoints()**

Endpoints-uri implicite

Metoda **`AddDefaultEndpoints()`**

In MSDN: "Adds service endpoints for all base addresses in each contract found in the service host with the default binding."

In WCF 3.x serviciul trebuia (in mod obligatoriu) sa fie configurat cu cel putin un endpoint. Incepand cu WCF 4.0, aceasta restrictie a fost eliminata. Daca serviciul nu are atasat un endpoint atunci se vor crea endpoints-uri in mod automat de catre sistem. Cand aplicatia ce gazduieste serviciul apeleaza metoda **Open**() pe instanta tipului **ServiceHost**, aceasta va construi descrierea interna a serviciului din fisierul de configurare al aplicatiei si din orice configurari explicite facute in cod, iar daca numarul endpoints-urilor configurate este zero va apela metoda **AddDefaultEndpoints**().

AddDefaultEndpoints() adauga unul sau mai multe endpoints-uri la descrierea serviciului, bazate pe adresa de baza a serviciului. Se adauga cate un endpoint implicit per adresa de baza pentru fiecare contract implementat de serviciu.

Exemplu. Daca serviciul implementeaza doua contracte de serviciu si configuram host-ul cu o singura adresa de baza, **AddDefaultEndpoints** va configura serviciul cu doua endpoints-uri implicite (unul pentru fiecare contract al serviciului). Daca serviciul implementeaza doua contracte de serviciu si host-ul este configurat cu doua adrese de baza (una pentru HTTP si alta pentru TCP), **AddDefaultEndpoints** va configura serviciul cu patru endpoints-uri implicite.

Observatie
Endpoints-urile implicite sunt adaugate numai atunci cand serviciul nu a fost configurat cu nici un endpoint.

Exemplu (MSDN)

```
// Definim doua contracte de serviciu: IHello si IGoodbye
// Fiecare contract prezinta cate o metoda: SayHello, respectiv SayGoodbye
[ServiceContract]
public interface IHello
{
    [OperationContract]
    void SayHello(string name);
}
[ServiceContract]
public interface IGoodbye
{
    [OperationContract]
    void SayGoodbye(string name);
}
// Serviciul implementeaza ambele contracte
public class GreetingService : IHello, IGoodbye {
    public void SayHello(string name)
    {
        Console.WriteLine("Hello {0}", name);
    }
    public void SayGoodbye(string name)
    {
        Console.WriteLine("Goodbye {0}", name);
```

```
    }
}
```

Codul din host poate fi:

```
class Server
{
    static void Main(string[] args)
    {
        // Host este configurat cu doua adrese de baza,
        // una pentru HTTP si una pentru TCP.

        ServiceHost host = new ServiceHost(typeof(GreetingService),
            new Uri("http://localhost:8080/greeting"),
            new Uri("net.tcp://localhost:8081/greeting"));
        host.Open();

        // Afisam ABC pentru fiecare endpoint din serviciu

        foreach (ServiceEndpoint se in host.Description.Endpoints)
            Console.WriteLine("A (address): {0}, B (binding): {1},
         C (Contract): {2}",
                se.Address, se.Binding.Name, se.Contract.Name);
        Console.WriteLine("Press <Enter> to stop the service.");
        Console.ReadLine();
        host.Close();
    }
}
```

Se vor afisa patru endpoints-uri: doua HTTP si doua TCP, pentru fiecare contract cate unul din fiecare adresa.

Mapare protocol implicit

WCF defineste o mapare implicita intre schema protocolului de transport (ex. http, net.tcp, net.pipe, etc.) si bindings-urile preconstruite in WCF. Maparea protocolului implicit se gaseste definita in fisierul *machine.config.comments* si arata astfel:

```
<system.serviceModel>
   <protocolMapping>
      <add scheme="http" binding="basicHttpBinding" bindingConfiguration="" />
      <add scheme="net.tcp" binding="netTcpBinding" bindingConfiguration=""/>
      <add scheme="net.pipe" binding="netNamedPipeBinding"
                bindingConfiguration=""/>
      <add scheme="net.msmq" binding="netMsmqBinding"
```

```
                bindingConfiguration=""/>
    </protocolMapping>
    ...
```

Aceste valori pot fi schimbate prin modificarea fisierului *machine.config* sau in fisierul de configurare al aplicatiei. In ex. de mai jos se ataseaza la schema **http** un binding **webHttpBinding**:

```
<configuration>
  <system.serviceModel>
    <protocolMapping>
      <add scheme="http" binding="webHttpBinding"/>
    </protocolMapping>
  </system.serviceModel>
</configuration>
```

Configurare implicita pentru binding : bindingConfiguration

Fiecare *binding* WCF vine cu o configurare implicita, configurare ce poate fi schimbata folosind fisiere de configurare sau din cod. In fisierul de configurare se foloseste atributul **bindingConfiguration** pentru elementul **endpoint**. In acest caz atributul **name** din elementul **binding** trebuie sa aiba aceeasi valoare ca cea furnizata de **bindingConfiguration**. De analizat exemplul de mai jos.

```
<configuration>
  <system.serviceModel>
    <bindings>
      <basicHttpBinding>
        <binding name="BasicWithMtom" messageEncoding="Mtom"/>
      </basicHttpBinding>
    </bindings>
    <services>
      <service name="GreetingService">
        <endpoint address="mtom"
                  binding="basicHttpBinding"
                  bindingConfiguration="BasicWithMtom"
                  contract="IHello"/>
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

In WCF 4 configurarea unui binding implicit se face prin omiterea atributului **name** din **<binding>** iar endpoint-urile care nu au precizat atributul **bindingConfiguration** vor folosi acest binding implicit.

Vezi si exemplul:

```
<configuration>
  <system.serviceModel>
    <bindings>
      <basicHttpBinding>
        <binding messageEncoding="Mtom"/> <!-- nu exista atributul name -->
      </basicHttpBinding>
    </bindings>
    <services>
      <service name="GreetingService">
        <endpoint address="mtom"
                  binding="basicHttpBinding"
                  contract="IHello"/>
      </service>
    </services>

  </system.serviceModel>
</configuration>
```

Configurare implicita pentru comportare - `behaviorConfiguration`

In WCF 3.x pentru configurarea comportarii serviciului, ***behavior***, se foloseste atributul **`behaviorConfiguration`**. In WCF 4, nu mai e nevoie de a folosi acest atribut. In fisierul de configurare am putea avea:

```
<configuration>
  <system.serviceModel>
    <behaviors>
      <serviceBehaviors>
        <behavior> <!-- nici un atribut name -->
          <serviceMetadata httpGetEnabled="true"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

In acest fel orice serviciu care nu are o configurare pentru *behavior* va folosi configurarea data mai sus, in acest caz **activarea publicarii metadatei** pentru serviciu.

Observatie

In cazul activarii publicarii metadatei pentru serviciu trebuie furnizat si endpoint-ul pentru contractul **`IMetadataExchange`**. Vezi mai jos in text - **`ServiceMetadataBehavior`**.

ServiceMetadataBehavior

Controleaza publicarea metadatei serviciului si informatia asociata.

Namespace: System.ServiceModel.Description
Assembly: System.ServiceModel (in System.ServiceModel.dll)

```
public class ServiceMetadataBehavior : IServiceBehavior
```

Adauga un obiect **ServiceMetadataBehavior** la colectia **ServiceDescription.::.Behaviors** sau elementul **<serviceMetadata>** intr-un fisier de configurare al aplicatiei pentru a permite sau nu publicarea metadatei serviciului. Oricum, adaugand *comportarea* la un serviciu nu este suficient pentru a permite publicarea metadatei.

Pentru a permite « regasirea metadatei », trebuie sa adaugam un endpoint la serviciul nostru in care contractul este **IMetadataExchange**. Endpoint-ul **IMetadaExchange** poate fi configurat ca orice alt endpoint.

Pentru a permite regasirea metadatei prin “HTTP GET” trebuie ca proprietatea

```
HttpGetEnabled = true;
```

Exemplu: Acest fisier de configurare ne arata modul de folosire pentru **ServiceMetadataBehavior** si permite suport pentru extragerea metadatei.

```
<configuration>
  <system.serviceModel>
    <services>
      <service
        name="Microsoft.WCF.Documentation.SampleService"
        behaviorConfiguration="metadataSupport">
        <host>
          <baseAddresses>
            <add baseAddress="http://localhost:8080/SampleService" />
          </baseAddresses>
        </host>
        <endpoint
          address=""
          binding="wsHttpBinding"
          contract="Microsoft.WCF.Documentation.ISampleService"/>
        <!-- Adds a WS-MetadataExchange endpoint at -->
        <!-- "http://localhost:8080/SampleService/mex" -->
        <endpoint
           address="mex"
           binding="mexHttpBinding"
           contract="IMetadataExchange"/>
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
      <behavior name="metadataSupport">
        <!-- Enables the IMetadataExchange endpoint in services that -->
        <!-- use "metadataSupport" in their behaviorConfiguration -->
        <!-- attribute. -->
```

```
        <!-- In addition, the httpGetEnabled and httpGetUrl -->
        <!-- attributes publish-->
        <!-- Service metadata for retrieval by HTTP/GET at the address -->
        <!-- "http://localhost:8080/SampleService?wsdl" -->
        <serviceMetadata httpGetEnabled="true" httpGetUrl=""/>
      </behavior>
    </serviceBehaviors>
  </behaviors>
  </system.serviceModel>
</configuration>
```

Proprietati - `ServiceMetadataBehavior` (MSDN)

Name	Description
HttpGetEnabled	Gets or sets a value that indicates whether to publish service metadata for retrieval using an HTTP/GET request.
HttpGetUrl	Gets or sets the location of metadata publication for HTTP/GET requests.
HttpsGetEnabled	Gets or sets a value that indicates whether to publish service metadata for retrieval using an HTTPS/GET request.
HttpsGetUrl	Gets or sets the location of metadata publication for HTTPS/GET requests.

Expunere endpoint-uri pentru metadata - exemplu

Exemplul urmator prezinta acelasi serviciu ca mai sus dar expunand endpoint-uri pentru metadata.
Expunerea metadatei se poate face folosind fisiere de configurare sau in mod programabil.
In exemplu, endpoint-ul pentru metadata este dat in mod programabil. Vezi comentariul din cod.

Metadata necesara pe partea de client este obtinuta cu **`svcutil.exe`**.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ServiceModel;

namespace WcfHost
{
    [ServiceContract]
    public interface IStockService
    {
        [OperationContract]
        double GetPrice(string _text);
    }

    public class StockService:IStockService
    {
        public double GetPrice(string _text)
        {
            return 12.13;
        }
    }

    public class MainHostService
    {
        public static void Main()
```

```
        {
            ServiceHost serviceHost = new ServiceHost(typeof(StockService),
                    new Uri("http://localhost:8000/StockWcf"));
            serviceHost.AddServiceEndpoint(typeof(IStockService),
                    new BasicHttpBinding(),"");

            // endpoint pentru metadata
            ServiceMetadataBehavior behavior =
                  new ServiceMetadataBehavior();
            behavior.HttpGetEnabled = true;
            serviceHost.Description.Behaviors.Add(behavior);
            serviceHost.AddServiceEndpoint(typeof(IMetadataExchange),
                    MetadataExchangeBindings.CreateMexHttpBinding(),
                    "mex");
            //

            serviceHost.Open();
            Console.WriteLine("Serviciu lansat. " +
                    "\nAsteptare conexiuni. " +
                    "\nApasati <Enter> pentru a inchide serviciul");
            Console.ReadLine();
            serviceHost.Close();
        }
    }
}
```

Contracte multiple si endpoint-uri intr-un serviciu

Reamintim ca un serviciu poate fi definit ca fiind o colectie de endpoint-uri.

endpoint = (A,B,C) ;

Adresa : ne arata unde este serviciul pe retea.
Binding-ul : cum accesam serviciul.
Contractul : este ceea ce expune endpoint-ul.

Exista o relatie de **1:n** intre contracte si endpoint-uri.

Un endpoint poate avea numai un contract, dar un contract poate fi referit de mai multe endpoint-uri.

Observatie
Agregarea interfetelor permite ca un singur contract sa expuna interfete multiple.

```
[ServiceContract]
interface IServiceA
{
   [OperationContract]
   string MA();
}

[ServiceContract]
interface IServiceB
{
```

```
   [OperationContract]
   string MB();
}

[ServiceContract]
interface IServiceC : IServiceA, IServiceB {}

public class ServiceABC : IServiceC
{
   string MA() { return "Iasi";}
   string MB() { return "Computer Science";}
}
```

iar in fisierul de configurare putem avea :

```
<configuration>
  <system.serviceModel>
    <services>
      <service
        name="ABC"
        behaviorConfiguration="metadataSupport">
        <host>
          <baseAddresses>
            <add baseAddress="http://localhost:8080/ServiceABC" />
          </baseAddresses>
        </host>
        <endpoint
          name=”sa”
          address="A"
          binding="wsHttpBinding"
          contract="namespace.IServiceA"
        />
        <endpoint
       name=”sb”
          address="B"
          binding="basicHttpBinding"
          contract="namespace.IServiceB"
        />
        <endpoint
       name=”sc”
          address="C"
          binding="netTcpBinding"
          contract="namespace.IServiceC"
        />
        <endpoint
       name=”scc”
          address="CC"
          binding="wsHttpBinding"
          contract="namespace.IServiceC"
        />
        <!-- Adds a WS-MetadataExchange endpoint at -->
        <!-- "http://localhost:8080/ServiceABC/mex" -->
        <endpoint
           address="mex"
           binding="mexHttpBinding"
           contract="IMetadataExchange"
        />
      </service>
    </services>
```

In caz ca acelasi contract este definit pe endpoint-uri multiple, trebuie sa completam atributul « ***name*** » in <**endpoint**> si sa apelam serviciul prin numele endpoint-ului :

```
ClientProxy cp = new ClientProxy("scc") ;
string result = cp.MB();
```

Mai mult, multiple endpoint-uri cu acelasi binding dar contracte diferite pot fi localizate la aceeasi adresa.

Expunand un contract prin mai multe endpoint-uri intr-un serviciu, il putem face disponibil prin mai multe binding-uri.

Atributele **[ServiceContract], [OperationContract], [ServiceBehavior], [MessageParameter]** contin ctori ce permit modificarea *numelui* contractului, serviciului, operatiei, parametrilor, etc. Pentru mai multe informatii consultati MSDN. Proprietatea publica in **ServiceContract** este **Name**.

Exemplu

```
[ServiceContract(Name="AltServiciu")]
```

Clasa ServiceContractAttribute

Indica daca o *interfata* sau *clasa* este un contract pentru serviciu intr-o aplicatie WCF.
Acest atribut se aplica la interfete sau clase.

Namespace: System.ServiceModel
Assembly: System.ServiceModel (in System.ServiceModel.dll)

Constructor

Name	Description
ServiceContractAttribute	Initializes a new instance of the ServiceContractAttribute class.

Proprietati

Name	Description
CallbackContract	Gets or sets the type of callback contract when the contract is a duplex contract.
ConfigurationName	Gets or sets the name used to locate the service in an application

	configuration file.
HasProtectionLevel	Gets a value that indicates whether the member has a protection level assigned.
Name	Gets or sets the name for the **<portType>** element in Web Services Description Language (WSDL).
Namespace	Gets or sets the namespace of the **<portType>** element in Web Services Description Language (WSDL).
ProtectionLevel	Specifies whether the binding for the contract must support the value of the ProtectionLevel property.
SessionMode	Gets or sets whether sessions are allowed, not allowed or required.

Exemplu - Implementarea unui serviciu WCF

O modalitate foarte simpla. Nu necesita fisiere de configurare. Pe partea de client se foloseste clasa **ChannelFactory**.

Observatie
Metadata este implementata pe partea de client cu **interfete**.

Se adauga referinta la **System.ServiceModel**.

Host

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ServiceModel;

namespace WcfHost
{
   // 1. Serviciul

     [ServiceContract]
     public interface IStockService
     {
         [OperationContract]
         double GetPrice(string _text);
     }

     public class StockService:IStockService
     {
```

```
        public double GetPrice(string _text)
        {
            return 12.13;
        }
    }
    // end serviciu

    // 2. Host pentru serviciu. Aplicatie consola.

    public class MainHostService
    {
        public static void Main()
        {
            ServiceHost serviceHost =
              new ServiceHost(typeof(StockService),
              new Uri("http://localhost:8000/StockWcf"));
            serviceHost.AddServiceEndpoint(typeof(IStockService),
                        new BasicHttpBinding(),"");

            serviceHost.Open();

            Console.WriteLine("Serviciu lansat. " +
                   "\nAsteptare conexiuni. " +
                   "\nApasati <Enter> pentru a inchide serviciul");
            Console.ReadLine();

            serviceHost.Close();
        }
    }
}
```

Clientul – foloseste interfata din serviciu

Se adauga referinta la **System.ServiceModel**.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using System.ServiceModel;

namespace WcfClient
{
    // 1. Metadata pentru serviciu
    [ServiceContract]
    public interface IStockService
    {
        [OperationContract]
        double GetPrice(string _text);
    }
    // end metadata

    class ClientWcf
    {
```

```
        static void Main(string[] args)
        {
            ChannelFactory<IStockService> channel =
              new ChannelFactory<IStockService>(
                 new BasicHttpBinding(),
                 new EndpointAddress("http://localhost:8000/StockWcf"));

             // Instanta tipului din serviciu
             IStockService wcfClient = channel.CreateChannel();

                double price = wcfClient.GetPrice("wcf");
                Console.WriteLine("Price = {0}", price);
                Console.ReadLine();
        }
    }
}
```

Explicatii. Pentru a se conecta la serviciu, clientul foloseste un obiect de tip **ChannelFactory** si metoda **CreateChannel**() din aceasta clasa. Serverul si clientul folosesc (in acest exemplu) un binding **Http**, **basicHttpBinding**, iar numele serviciului este *StockWcf*. **Portul** pe care se asteapta mesajele este **8000**.

In MSDN **ChannelFactory** este descrisa astfel :

```
// Summary:
//     A factory that creates channels of different types
// that are used by clients to send messages to variously
// configured service endpoints.
//
// Type parameters:
//   TChannel:
//     The type of channel produced by the channel factory.
//     This type must be either
//     System.ServiceModel.Channels.IOutputChannel or
//     System.ServiceModel.Channels.IRequestChannel.

public class ChannelFactory<TChannel> : ChannelFactory,
              IChannelFactory<TChannel>,
              IChannelFactory,
              ICommunicationObject

        // Summary:
        //     Initializes a new instance of the
        //        System.ServiceModel.ChannelFactory<TChannel>
        //     class with a specified binding and endpoint address.
        //
        // Parameters:
        //   binding:
        //     The System.ServiceModel.Channels.Binding used to
        //     configure the endpoint.
        //
        //   remoteAddress:
        //     The System.ServiceModel.EndpointAddress that provides
        //     the location of the service.
```

```
        //
        // Exceptions:
        //   System.ArgumentNullException:
        //     The binding is null.
        public ChannelFactory(Binding binding, EndpointAddress remoteAddress);
```

Obtinere metadata pentru client.

1. Lansam serviciul (codul de mai sus).
2. In *command prompt* din Visual Studio scriem urmatoarea linie de comanda :

```
D:\Program Files\Microsoft Visual Studio 10.0\VC>cd \

D:\>svcutil http://localhost:8000/StockWcf/mex -config:app.config
                  -out:proxy.cs

Microsoft (R) Service Model Metadata Tool
[Microsoft (R) Windows (R) Communication Foundation, Version 4.0.30319.1]
Copyright (c) Microsoft Corporation.  All rights reserved.

Attempting to download metadata from 'http://localhost:8000/StockWcf/mex'
using
WS-Metadata Exchange or DISCO.
Generating files...
D:\proxy.cs
D:\app.config
```

Copiem aceste fisiere (*proxy.cs* si *app.config*) in directorul unde construim clientul (proiectul clientului) si le adaugam la proiect. Nu uitam sa adaugam referinta la **System.ServiceModel**. Vizualizam *proxy.cs* si vom observa ca este creata clasa **StockServiceClient**. Codul din client in acest caz este:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ServiceModel;

namespace WcfClientMex
{
    class ClientWcfMex
    {
        static void Main(string[] args)
        {
         // clasa definita in proxy.cs si creata de utilitarul svcutil.exe
            StockServiceClient proxy = new StockServiceClient();

         // apel metoda din serviciu
            double dd = proxy.GetPrice("Iasi");

            Console.WriteLine("Client start...");
            Console.WriteLine("Valoare obtinuta din GetPrice = {0}", dd);

            Console.ReadLine();
        }
    }
}
```

Observatie
In fisierul de configurare, *app.config*, nu s-a facut nici o modificare.
Continutul acestuia este:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <system.serviceModel>
        <bindings>
            <basicHttpBinding>
                <binding name="BasicHttpBinding_IStockService"
                closeTimeout="00:01:00"
                   openTimeout="00:01:00" receiveTimeout="00:10:00"
                sendTimeout="00:01:00"
                   allowCookies="false" bypassProxyOnLocal="false"
                hostNameComparisonMode="StrongWildcard"
                   maxBufferSize="65536" maxBufferPoolSize="524288"
                maxReceivedMessageSize="65536"
                   messageEncoding="Text" textEncoding="utf-8"
                transferMode="Buffered"
                   useDefaultWebProxy="true">
                   <readerQuotas maxDepth="32" maxStringContentLength="8192"
                       maxArrayLength="16384"
                         maxBytesPerRead="4096" maxNameTableCharCount="16384"
                   />
                   <security mode="None">
                         <transport clientCredentialType="None"
                               proxyCredentialType="None"
                            realm="" />
                         <message clientCredentialType="UserName"
                      algorithmSuite="Default" />
                   </security>
                </binding>
            </basicHttpBinding>
        </bindings>
        <client>
            <endpoint address="http://localhost:8000/StockWcf"
               binding="basicHttpBinding"
                     bindingConfiguration="BasicHttpBinding_IStockService"
               contract="IStockService"
                     name="BasicHttpBinding_IStockService" />
        </client>
    </system.serviceModel>
</configuration>
```

Clasa OperationContractAttribute - MSDN

Reamintim:
Contractele pentru serviciu se adnoteaza cu atributul [**ServiceContract**].
Operatiile pentru serviciu se adnoteaza cu atributul [**OperationContract**].

Atributul se aplica metodelor expuse de serviciu, ceea ce semnifica ca metoda in cauza, cunoscuta si sub numele de operatie a contractului, poate fi apelata de client. Acest atribut controleaza structura operatiei si valorile publicate in metadata.

Proprietatile cele mai importante sunt:

- **`Action`** : specifica actiunea ce identifica in mod unic aceasta operatie.
- **`AsyncPattern`** : operatia este implementata sau poate fi apelata in mod asincron folosind metodele perechi Begin / End.
- **`IsOneWay`** : operatia consta dintr-un singur mesaj de intrare, nu exista raspuns si nu se asteapta raspuns.
- **`IsInitiating`** : specifica daca aceasta operatie poate fi operatia initiala intr-o sesiune.
- **`ProtectionLevel`** : nivelul de securitate al mesajului pe care o operatie il cere la runtime. Mesajul poate fi semnat (*signed*), criptat (*encrypted*) sau semnat si criptat (*signed and encrypted*).
- **`ReplyAction`** : specifica actiunea mesajului "reply" pentru operatie.

// MSDN
The **IsOneWay** property indicates that a method does not return any value at all, including an empty underlying response message. This type of method is useful for notifications or event-style communication. Methods of this kind cannot return a reply message so the method's declaration must return **void.**

The **Action** and **ReplyAction** properties can be used not only to modify the default action of SOAP messages but also to create handlers for unrecognized messages or to disable adding actions for direct message programming. Use the **IsInitiating** property to prevent clients from calling a particular service operation prior to other operations. Use the **IsTerminating** property to have close the channel after clients call a particular service operation. For more information, see **Using Sessions.**

The **ProtectionLevel** property enables you to specify on the operation contract whether the operation messages are signed, encrypted, or signed and encrypted. If a binding cannot provide the security level required by the contract, an exception is thrown at run time. For more information, see **ProtectionLevel** and **Understanding Protection Level.**
// End - MSDN

Un contract de serviciu descrie interfata la operatiile implementate de un endpoint.
Formatele mesajelor sunt descrise de contractele pentru date si mesaje.

Tipuri de operatii

Operatie cerere-raspuns sincrona

In acest caz operatia este blocanta, adica se asteapta terminarea executiei operatiei indiferent daca operatia returneaza sau nu.

Binding-uri folosite:
basicHttpBinding
netTcpBinding

Operatie cerere-raspuns asincrona

In obtinerea metadatei cu svcutil.exe se foloseste switch **/async**

Bazat pe interfata **IAsyncResult** si metode de tip ***Begin...*** si ***End...***.

```
// Summary:
//     Represents the status of an asynchronous operation.
[ComVisible(true)]
public interface IAsyncResult
{
        object AsyncState { get; }
//
// Summary:
//     Gets a System.Threading.WaitHandle that is used to wait for an
//  asynchronous operation to complete.
//
// Returns:
//     A System.Threading.WaitHandle that is used to wait for
// an asynchronous operation to complete.
        WaitHandle AsyncWaitHandle { get; }

     bool CompletedSynchronously { get; }
        bool IsCompleted { get; }
    }
}
```

Metodele ***Begin...*** contin de obicei un parametru (delegate) ce indica metoda ce va fi apelata cand operatia s-a terminat.

Operatie One-Way : [OperationContract(IsOneWay=true)]

In general operatia este neblocanta.
Dupa ce clientul initiaza apelul, WCF genereaza un mesaj de tip cerere (*request message*) care nu e corelat cu un raspuns catre client. Operatia *one-way* nu este acelasi lucru cu o operatie asincrona. Cererea clientului (apelul metodei one-way) poate sa nu fie prelucrata imediat si ca atare va fi pusa intr-o coada de

asteptare. Daca serviciul nu reuseste sa puna cererea in coada de asteptare, clientul este blocat. Clientul va fi deblocat cand se va reusi plasarea cererii in coada de asteptare.

Toate binding-urile din WCF suporta operatii one-way (Juval Lowy).

Declararea unei operatii *one-way* se face ca in exemplul de mai jos:

```
[ServiceContract]
interface IServiceA
{
   [OperationContract(IsOneWay=true)]
   void OperatieOneWay();
}
```

Obsevatie
Opeartia ***one-way*** trebuie sa returneze void.

Operatie Duplex

Comunicarea de tip cerere-raspuns este initiata de client. Clientul trimite un mesaj la serviciu (*request message*) si apoi serviciul trimite un mesaj de raspuns la client. Poate fi folosit pattern-ul sincron sau asincron.

Pentru operatiile « **duplex** » protocolul de comunicare este acelasi in ambele directii.

WCF permite comunicarea bidirectionala prin intermediul unui contract de serviciu duplex.
Un contract de serviciu duplex implementeaza pattern-ul mesaj duplex in care mesajele nesolicitate pot fi trimise in orice directie dupa ce s-a stabilit canalul de comunicatie.

Operatiile peste un canal duplex pot fi ***request-reply*** sau ***one-way***.

In acest caz *atat clientul cat si serverul* au nevoie de definirea unei *adrese*, a unui *binding* si a unui *contract* (ABC).

Pentru a facilita transmiterea mesajelor de la serviciu catre client, WCF poate crea canale aditionale. Daca canalul initial nu poate suporta comunicarea bidirectionala, atunci WCF creaza un al doilea canal folosind acelasi protocol specificat in endpoint-ul serviciului si facand un protocol simetric in ambele directii. In concluzie WCF poate crea unul sau doua canale pentru a permite comunicarea bidirectionala.

`NetNamedPipeBinding` si **`NetTcpBinding`** suporta comunicarea bidirectionala. Binding-urile care au in nume cuvantul *dual* suporta comunicarea bidirectionala – **`wsDualHttpBinding`**.

`HTTP` nu poate fi utilizat pentru *callback*, deci binding-urile **`basicHttpBinding`** si **`wsHttpBinding`** nu pot fi utilizate in comunicarea bidirectionala.

Un contract de serviciu poate avea *cel mult* un contract *callback*. Odata ce contractul *callback* a fost definit, cerinta principala a clientilor este de a suporta callback si de a furniza serviciului endpoint-urile pentru callback in fiecare apel.

Atributul [**ServiceContract**] ofera proprietatea **CallbackContract** de tipul **Type**, pentru a descrie o asemenea functionalitate.

```
interface IMessageCallback
{
    [OperationContract(IsOneWay = true)]
    void OnMessageAdded(string message, DateTime timestamp);
}

[ServiceContract(CallbackContract = typeof(IMessageCallback))]
public interface IMessage
{
    [OperationContract]
    void AddMessage(string message);

     // cod
}
```

Setare callback pe partea de client

Clase importante:

- **InstanceContext** – folosita de client.
- **OperationContext** – folosita de serviciu. Metoda importanta aici este **GetCallbackChannel**.

Clientul va gazdui obiectul *callback* si va expune un endpoint pentru *callback*.

Clasa folosita aici este **InstanceContext** ce are un ctor de forma:

```
public InstanceContext(object implementation) {...}
```

Toti clientii ce folosesc *callback* vor construi obiectul *callback* (instanta clasei ce defineste obiectul) si un context pentru acesta folosind clasa **InstanceContext** :

```
class MyCallback : IMessageCallback
{
   public void OnCallback(){...}
      IMessageCallback callback = new MyCallback() ;
      InstanceContext context = new InstanceContext(callback);
}
```

Cand se interactioneaza cu un endpoint al serviciului al carui contract defineste un contract callback, clientul trebuie sa foloseasca un proxy ce va seta comunicarea bidirectionala si sa transfere referinta endpoint-ului callback catre serviciu.

Clasa **OperationContext** este folosita pe partea de serviciu pentru a accesa referinta callback folosind metoda **GetCallbackChannel**.

Serviciul poate extrage referinta la obiectul callback din contextul operatiei si o poate memora pentru a o folosi la apelul callback.

```
// extragere referinta – cod pe partea de serviciu
```

```
 IMessageCallback callback =
 OperationContext.Current.GetCallbackChannel<IMessageCallback>();

// memorare intr-o lista - cod pe partea de serviciu
if (!subscribers.Contains(callback))
    subscribers.Add(callback);
    return true;

// apel viitor - cod pe partea de serviciu
  subscribers.ForEach(delegate(IMessageCallback callback)
        {
             if (((ICommunicationObject)callback).State ==
                              CommunicationState.Opened)
             {
                   callback.OnMessageAdded(message, DateTime.Now);
             }
             else
             {
                   subscribers.Remove(callback);
             }
         });
```

Proxy pentru client va contine o clasa derivata din **DuplexClientBase** :

```
public partial class MessageClient :
System.ServiceModel.DuplexClientBase<IMessage>, IMessage
{
  // cod omis
}
```

Perechi operatii One-way versus Duplex

Exista doua scenarii.

1. Implementam pe partea de serviciu operatie *one-way* si pe partea de client un serviciu ce are operatie *one-way*. In acest caz ambele aplicatii sunt host pentru diferite servicii si pot comunica prin endpoint-urile definite. In acest caz protocoalele de comunicatie pot fi diferite.

2. Implementam pe partea de serviciu o operatie duplex caz in care clientul nu este in mod explicit un host pentru alt serviciu.

Adresa, binding-ul si contractul ce defineste un endpoint pe partea de client sunt implementate de “channel factory” cand se initiaza comunicarea duplex de catre client.

Exemplu: Implementare contract de serviciu duplex

Un contract duplex contine specificatiile pentru serviciu si endpoint-urile clientului. In acest tip de contract, portiuni ale contractului pe partea de serviciu sunt implementate de client.

Exemplu complet este compus din trei proiecte.

1. Proiect de tip Class Library ce va contine tipurile expuse in server.
   a. Interfetele si clasa ce implementeaza aceste interfete
   b. Fisierul de configurare
2. Proiect de tip Windows Application ce va contine serverul
   a. Fisierul de configurare
3. Clientul
   a. Proxy – ce contine clasa MessageClient
   b. Fisierul de configurare

Incepem prin a descrie serviciul. Deoarece serverul va apela metode de pe partea clientului (notificare client) clasa ce implementeaza serviciul va trebui sa mentina o lista a clientilor ce vor fi notificati, lista ce va contine in fapt metoda callback ce va trebui apelata.

Serviciul se defineste astfel.

1. Interfata ce defineste metoda callback. Aceasta interfata este in .dll de pe server dar este implementata in client.

Observatie
Interfata **IMessageCallback** nu este adnotata cu atributul [**ServiceContract**], in schimb metoda este adnotata cu atributul **[OperationContract(IsOneWay=true)]** deci nu se asteapta raspuns de la aceasta metoda.

```
// Fisierul IMessageCallback.cs
namespace WCFCallbacks
{
    using System;
    using System.ServiceModel;

    interface IMessageCallback
    {
        [OperationContract(IsOneWay = true)]
        void OnMessageAdded(string message, DateTime timestamp);
    }
}
```

2. *Serviciul definit prin interfata **IMessage***

```
// Fisierul IMessage.cs
namespace WCFCallbacks
{
    using System.ServiceModel;
    [ServiceContract(CallbackContract = typeof(IMessageCallback))]
    public interface IMessage
    {
        [OperationContract]
        void AddMessage(string message);

        [OperationContract]
        bool Subscribe();

        [OperationContract]
        bool Unsubscribe();
    }
```

```
}
```

Metodele *Subscribe* si *Unsubscribe* gestioneaza o colectie de metode callback apelate pe partea clientului. Tipul colectiei este **IMessageCallback**.
Metoda *AddMessage* va contine codul pentru apelul metodei callback.

Clasa ce implementeaza interfata **IMessage**, adica serviciul.

```
// Fisierul MessageService.cs
namespace WCFCallbacks
{
    using System;
    using System.Collections.Generic;
    using System.ServiceModel;

    public class MessageService : IMessage
    {
        private static readonly List<IMessageCallback>
         subscribers = new List<IMessageCallback>();

        public void AddMessage(string message)
        {
            // aici se face apelul la metoda callback

            subscribers.ForEach(delegate(IMessageCallback callback)
            {
                if (((ICommunicationObject)callback).State ==
                            CommunicationState.Opened)
                {
                    callback.OnMessageAdded(message, DateTime.Now);
                }
                else
                {
                    subscribers.Remove(callback);
                }
            });
        }

        public bool Subscribe()
        {
            try
            {
                IMessageCallback callback =
                     OperationContext.Current.
                     GetCallbackChannel<IMessageCallback>();

    // Inregistrez "tipul" ce implementeaza IMessageCallback
    // Acest tip este pe partea de client

                if (!subscribers.Contains(callback))
                    subscribers.Add(callback);
                return true;
            }
            catch
            {
                return false;
            }
        }
```

```
        public bool Unsubscribe()
        {
            try
            {
                IMessageCallback callback =
                    OperationContext.Current.
                    GetCallbackChannel<IMessageCallback>();
                if (subscribers.Contains(callback))
                    subscribers.Remove(callback);
                return true;
            }
            catch
            {
                return false;
            }
        }
    }
}
```

Explicatii

Stabilirea metodei callback se face cu secventa de cod:

```
IMessageCallback callback =
   OperationContext.Current.GetCallbackChannel<IMessageCallback>();
```

apelata in *Subscribe* si *Unsubscribe*. Descrierea din SDK este data mai jos.

```
//     Provides access to the execution context of a service method.
public sealed class OperationContext : IExtensibleObject<OperationContext>

//     Gets or sets the execution context for the current thread.
//
// Returns:
//     The System.ServiceModel.OperationContext that represents
//     the messaging and execution context of the current method.
        public static OperationContext Current { get; set; }

//     Gets a channel to the client instance that called the
//     current operation.
//
// Type parameters:
//   T: The type of channel used to call back to the client.
//
// Returns:
//     A channel to the client instance that called the operation
//     of the type specified
//     in the System.ServiceModel.ServiceContractAttribute.CallbackContract
//     property.
        public T GetCallbackChannel<T>();
```

Apelul metodei callback se face in ***AddMessage***. Se verifica mai intai starea canalului de comunicatie si daca acesta este in starea « deschis » se apeleaza metoda ***OnMessageAdded*** pe tipul pastrat in colectie, adica tipul ce implementeaza interfata ***IMessageCallback*** de pe partea clientului, tip inregistrat cu metoda ***Subscribe***.

```
//     Defines the contract for the basic state machine for
//     all communication-oriented objects in the system,
```

```
    //      including
    //        channels,
    //        the channel managers,
    //        factories,
    //        listeners,
    //        and dispatchers,
    //        and service hosts.
    public interface ICommunicationObject
```

3. Implementare server. Cod...

a. Se adauga referinta la biblioteca construita anterior.

```
namespace WCFServiceHost
{
    using System;
    using System.ServiceModel;
    using System.ServiceModel.Description;

    class WCFServer
    {
        static void Main(string[] args)
        {
            ServiceHost myService =
               new ServiceHost(typeof(WCFCallbacks.MessageService),
                     new
Uri("net.tcp://localhost:9191/WCFCallbacks/Message/"));

            ServiceDescription serviceDesciption = myService.Description;

         // enumerare endpoint-uir din serviciu - informativ
            foreach (ServiceEndpoint endpoint in serviceDesciption.Endpoints)
            {
                ConsoleColor oldColour = Console.ForegroundColor;
                Console.ForegroundColor = ConsoleColor.Yellow;
                Console.WriteLine("Endpoint - address:  {0}",
                                  endpoint.Address);
                Console.WriteLine("          - binding name:\t\t{0}",
                      endpoint.Binding.Name);
                Console.WriteLine("          - contract name:\t\t{0}",
                     endpoint.Contract.Name);
                Console.WriteLine();
                Console.ForegroundColor = oldColour;
            }

            myService.Open();
            Console.WriteLine("Press enter to stop.");
            Console.ReadKey();

            if (myService.State != CommunicationState.Closed)
                myService.Close();
        }
    }
}
```

iar fisierul de configurare pentru server este:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.web>
    <compilation debug="true" />
  </system.web>
  <system.serviceModel>
    <services>
      <service name="WCFCallbacks.MessageService"
               behaviorConfiguration="WCFCallbacks.MessageBehavior">

        <endpoint address ="net.tcp://localhost:9191/WCFCallbacks/Message"
               binding="netTcpBinding"
               contract="WCFCallbacks.IMessage">
          <identity>
            <dns value="localhost"/>
          </identity>
        </endpoint>
        <endpoint address="mex"
               binding="mexTcpBinding"
               contract="IMetadataExchange"/>
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
        <behavior name="WCFCallbacks.MessageBehavior">
          <serviceMetadata />
          <serviceDebug includeExceptionDetailInFaults="False" />
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

Observati cum e configurat endpoint-ul aici. Nu se specifica nimic despre **IMessageCallback**.

4. Implementare client. Aplicatie tip consola.

a. Se lanseaza serverul si se obtine metadata si fisierul de configurare cu **svcutil.exe**.
b. **svcutil net.tcp://localhost:9191/WCFCallbacks/Message**
-config:App.config –out :poxy.cs
c. Se copie cele doua fisiere in directorul proiectului pentru client si se adauga apoi la proiect.
d. Adaugare referinta la **System.ServiceModel**

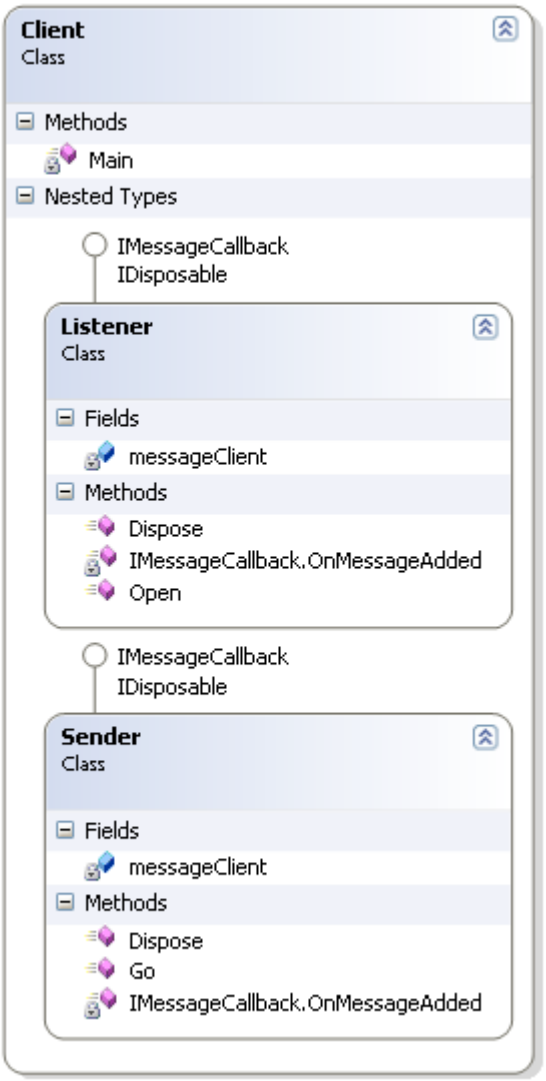
Proxy generat contine (am eliminat atribute si implementarile pentru a fi lizibil) :

```
[System.ServiceModel.ServiceContractAttribute(ConfigurationName="IMessage",
CallbackContract=typeof(IMessageCallback))]
public interface IMessage
{
    // cod omis
}

public interface IMessageCallback
{

    [System.ServiceModel.OperationContractAttribute(IsOneWay=true,
         Action="http://tempuri.org/IMessage/OnMessageAdded")]
    void OnMessageAdded(string message, System.DateTime timestamp);
```

```
    }

    public interface IMessageChannel : IMessage,
System.ServiceModel.IClientChannel
    {
    }

    public partial class MessageClient :
System.ServiceModel.DuplexClientBase<IMessage>, IMessage
    {
      // cod omis
    }
```

Observatie
`IMessageCallback` apare in proxy dar nu este adnotat cu **`ServiceContract`** in schimb apare in atributul de la interfata **`IMessage`** (la fel cum l-am definit in class library).

Diagrama clasei *Client* este urmatoarea:



Observatie
Urmariti comentariile din cod pentru a intelege mai bine ce se intampla.

```
// Fisierul Client.cs
namespace MessageSender
{
    using System;
    using System.ServiceModel;

    /// <summary>
    /// Clientul implementeaza metoda ce va fi apelata din serviciu
    /// in cadrul unui contract duplex.
    ///
    /// Se folosesc doua clase : Listner si Sender in ideea de a
    /// separa codul si actiunile de trimitere a mesajelor,
    /// de actiunea de executie a metodei callback din client.
    ///
    /// Se poate realiza callback si folosind numai una
    /// din aceste clase. Atentie la metoda Subscribe in acest caz.
    ///
    /// Tipul MessageClient este proxy pentru obiectul din server
    ///
    /// </summary>
    class Client
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Press enter when the server is running.");
            Console.ReadKey();

            // Subscribe pe server. Instiintez serviciul pe
            // ce obiect sa apeleze callback.
            // Se va apela callback pe metoda OnMessageAdded
            // din clasa Listner.

            Listener listener = new Listener();
            listener.Open();

            // Apelez metoda AddMessage din serviciu
            Sender sender = new Sender();
            sender.Go();

           sender.Dispose();
           listener.Dispose();

            Console.WriteLine("Done, press enter to exit");
            Console.ReadKey();
        }

        /// <summary>
        /// Obiecte din aceasta clasa apeleaza metoda AddMessage din serviciu.
        /// Initiatorul conversatiei
        /// </summary>
        [CallbackBehavior(ConcurrencyMode = ConcurrencyMode.Reentrant)]
        class Sender : IMessageCallback, IDisposable
        {
            // Creez obiect de pe server
            private MessageClient messageClient;

            public void Go()
            {
```

```
            // Reprezinta informatia de context pentru instanta
            // serviciului.
            // Cu acest context din proxy generat - clasa MessageClient -
            // se construieste un obiect folosind un ctor cu doi parametri
            // In app.config din client veti gasi
           /*
           <endpoint
              address="net.tcp://localhost:9191/WCFCallbacks/Message"
              binding="netTcpBinding"
              bindingConfiguration="NetTcpBinding_IMessage"
              contract="IMessage" name="NetTcpBinding_IMessage"
           >
          */
            // al doilea parametru fiind bindingConfiguration
            // context e necesar in procesul callback

            InstanceContext context = new InstanceContext(this);

            messageClient = new MessageClient(context,
                        "NetTcpBinding_IMessage");
            // atentie: e case sensitive "NetTcpBinding"
            //
            // Daca decomentam linia ce urmeaza atunci se va executa
            // IMessageCallback.OnMessageAdded(...) implementata
            // in aceasta clasa.
            // messageClient.Subscribe();
            //

            for (int i = 0; i < 5; i++)
            {
                string message = string.Format("message #{0}", i);
                Console.WriteLine(">>> Sending -> " + message);
                messageClient.AddMessage(message);
            }

        }

        void IMessageCallback.OnMessageAdded(string message,
                  DateTime timestamp)
        {
            Console.WriteLine("<<< [Sender OnMessageAdded] => Recieved {0}
                        with a timestamp of {1}",
                         message, timestamp);
        }

        public void Dispose()
        {
            messageClient.Close();
        }
    }

    /// <summary>
    /// Aceasta clasa e folosita pentru a indica metoda din client
    /// ce va fi apelata din cadrul serviciului.
    /// Observati ca se implementeaza interfata IMessageCallback.
    /// </summary>
    [CallbackBehavior(ConcurrencyMode = ConcurrencyMode.Reentrant)]
    class Listener : IMessageCallback, IDisposable
    {
        private MessageClient messageClient;
```

```
            public void Open()
            {
                // Reprezinta informatia de context pentru instanta
                // serviciului.
                // Cu acest context din proxy generat - clasa MessageClient -
                // se construieste un obiect folosind un ctor cu doi parametri
                // In app.config din client veti gasi
                /*
                 <endpoint
                      address="net.tcp://localhost:9191/WCFCallbacks/Message"
                      binding="netTcpBinding"
                      bindingConfiguration="NetTcpBinding_IMessage"
                      contract="IMessage" name="NetTcpBinding_IMessage">
               */
                // al doilea parametru fiind bindingConfiguration
                // context e necesar in procesul callback

                InstanceContext context = new InstanceContext(this);
                messageClient = new MessageClient(context,
                      "NetTcpBinding_IMessage");

                messageClient.Subscribe();
            }

            void IMessageCallback.OnMessageAdded(string message,
                      DateTime timestamp)
            {
                Console.WriteLine("<<< [Listner OnMessageAdded] >>>
                      Recieved {0} with a timestamp of {1}",
                      message, timestamp);
            }

            public void Dispose()
            {
                messageClient.Unsubscribe();
                messageClient.Close();
            }
        }
    }
}
```

si fisierul de configurare complet (client)

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.serviceModel>
    <bindings>
      <netTcpBinding>
        <binding      name="NetTcpBinding_IMessage"
               closeTimeout="00:01:00"
               openTimeout="00:01:00"
               receiveTimeout="00:10:00"
               sendTimeout="00:01:00"
               transactionFlow="false"
               transferMode="Buffered"
               transactionProtocol="OleTransactions"
               hostNameComparisonMode="StrongWildcard"
               listenBacklog="10"
               maxBufferPoolSize="524288"
```

```
                maxBufferSize="65536"
                   maxConnections="10"
                maxReceivedMessageSize="65536">
            <readerQuotas maxDepth="32"
                          maxStringContentLength="8192"
                          maxArrayLength="16384"
                          maxBytesPerRead="4096"
                          maxNameTableCharCount="16384" />
            <reliableSession       ordered="true"
                             inactivityTimeout="00:10:00"
                             enabled="false" />
            <security mode="Transport">
              <transport     clientCredentialType="Windows"
                        protectionLevel="EncryptAndSign" />
              <message clientCredentialType="Windows" />
            </security>
          </binding>
        </netTcpBinding>
      </bindings>


       <client>
        <endpoint address="net.tcp://localhost:9191/WCFCallbacks/Message"
                  binding="netTcpBinding"
                  bindingConfiguration="NetTcpBinding_IMessage"
                  contract="IMessage"
                  name="NetTcpBinding_IMessage">
          <identity>
            <dns value="localhost" />
          </identity>
        </endpoint>
      </client>
    </system.serviceModel>
  </configuration>
```

Comportare WCF

Comportarea unui serviciu afecteaza operatia la runtime si este definita de o serie de clase din .NET. Aceste clase sunt invocate cand se lanseaza WCF pe partea de server cat si pe partea de client.

Exista trei tipuri primare pentru comportari.

- **Service behaviors** : ruleaza la nivel de serviciu si au acces la toate endpoint-urile.
- **Endpoint behaviors** : sunt pentru endpoint-urile serviciului.
- **Operation behaviors** : actioneaza la nivelul operatiei si sunt folosite in special pentru manipularea serializarii, fluxului tranzactiilor si gestionarea parametrilor pentru o operatie a serviciului.

Alaturi de aceste trei comportari, WCF defineste comportare pentru **callback**, ce gestioneaza endpoint-urile create pe client pentru comunicarea bidirectionala.

Cum este initializat runtime-ul pentru WCF?

Pe partea de client exista **`ChannelFactory`**, iar pe partea de server **`ServiceHost`**.

Ambele clase executa functii similare :

1. Accepta un tip .NET ca intrare si ii citeste atributele.

2. Incarca informatia din fisierele *app.config* sau *web.config*.

- Pe partea de client, **`ChannelFactory`** cauta informatii despre *binding* ;
- Pe partea de server, **`ServiceHost`** cauta informatii despre *contract* si *binding*.

3. Se construieste structura mediului runtime, **`ServiceDescription`**.

4. Start comunicare.

- Pe partea de client, **`ChannelFactory`** foloseste canalul pentru a se conecta la serviciu.
- Pe partea de server, **`ServiceHost`** deschide canalul si asculta pentru mesaje.

In pasul **1**, informatia pentru comportare este definita prin atribute, ex.
`[ServiceBehavior(TranscationTimeout= "00:00:20")].`

In pasul **2**, informatia pentru comportare este definita in fisierul de configurare *app.config*, exemplu
`<transactionTimeout="00:00:20">.`

In timpul pasului **3**, clasele **`ChannelFactory`** si **`ServiceHost`** construiesc runtime-ul pentru WCF si insereaza comportarile gasite in pasii 1 si 2. Tot acum pot fi adaugate manual (prin cod) la modelul serviciului anumite comportari, ex.

```
Endpoint.Behaviors.Add(new myBehavior()).
```

Comportarile pot opera pe date inainte ca acestea sa fie transmise sau dupa ce au fost receptionate (asa numitele pre si post procesari).

Pe partea de client, comportarile pot fi folosite pentru a executa urmatoarele :

- **Inspectie parametri.** Inspecteaza si/sau schimba data in reprezentarea .NET inainte ca aceasta sa fi fost convertita in XML.
- **Formatare mesaj.** Inspecteaza si/sau schimba data in timpul conversiei intre tipurile .NET si XML.
- **Inspectie mesaj.** Inspecteaza si/sau schimba data in reprezentarea XML inainte ca aceasta sa fie convertita in tipuri .NET.

Pe partea de server, comportarile pot fi folosite pentru urmatoarele scenarii :

- **Selectie operatie**. La nivel de serviciu, inspecteaza mesajele ce vin si determina ce operatie va fi apelata.
- **Invocare operatie**. La nivel de operatie, invoca metoda din clasa.

Comportare serviciu - Concurenta si instantiere

Cateva definitii.

Concurenta este o masura pentru a evidentia cate task-uri pot fi executate simultan, prin task intelegand cerere, job-uri, tranzactii, etc.

Timpul de executie masoara timpul necesar unei operatii pentru a se executa (unitate de masura : milisecunde, secunde, etc.).

Throughput : masoara cate task-uri sunt executate complet intr-un timp fixat si se raporteaza ca *task/time* (cereri/secunda, tranzactii/minut, etc.). *Throughput* este functie de (depine de) concurenta si timp de executie. *Throughput* poate creste fie prin micsorarea timpului de executie (modificare algoritmi, alocare de resurse) fie prin cresterea concurentei (executie paralela).

Controlarea concurentei in WCF poate fi facuta cu ajutorul urmatoarelor proprietati :

- **`InstanceContextMode`**
- **`ConcurrencyMode.`**

`InstanceContextMode` specifica modul de creare al instantelor la nivel de server. Valorile posibile sunt :

- **`Single`** : O instanta a clasei serviciului serveste toate cererile ce vin de la clienti. Acest mod implementeaza un singleton.
- **`PerCall`** : O instanta a clasei serviciului este creata pentru fiecare cerere.
- **`PerSession`** : O instanta a clasei serviciului este creata pentru fiecare sesiune a clientului. Cand folosim canale fara sesiune, toate serviciile se comporta ca **`PerCall`** chiar daca am specificat **`PerSession`**.

Valoare implicita PerSession.

Exemple de declaratii

```
[ServiceContract]
interface IStockService
{ // ...
}

[ServiceBehavior(InstanceContextMode=InstanceContextMode.PerSession)]
public class StockService : IStockService
{ // implementari metode
}
```

`ConcurrencyMode` : (comportare serviciu) controleaza concurenta firului din interiorul unei instante a serviciului, mai exact daca o instanta a obiectului **callback** suporta un fir sau fire multiple ce se executa in mod concurent si in cazul cand suporta un singur fir controleaza daca este suportata reentranta.

Valorile posibile sunt :

- **`Single`**. Numai un fir la un moment dat poate accesa clasa serviciului. Este modul cel mai sigur de executie.
- **`Reentrant`**. Numai un fir la un moment dat poate accesa clasa serviciului, dar firul poate parasi clasa si poate reveni mai tarziu pentru a-si continua executia.
- **`Multiple`**. Fire multiple pot accesa clasa serviciului in mod simultan. Clasa trebuie sa fie construita thread-safe.

Valoarea implicita este **ConcurrencyMode.Single**, ceea ce inseamna ca WCF va executa numai un fir per instanta clasei serviciului.

Despre *reentrant* (Wikipedia) : Metoda nu e reentranta.

```
int t; // aici e problema cu variabila globala
void swap(int *x, int *y)
{
        t = *x;
        *x = *y;
        // hardware interrupt might invoke isr() here!
        *y = t;
}

void isr()
{
        int x = 1, y = 2;
        swap(&x, &y);
}
```

Metoda este reentranta:

```
int t; // variabila globala
void swap(int *x, int *y)
{
        int s;  // variabila locala
        s = t;  // save global variable
        t = *x;
        *x = *y;
        // hardware interrupt might invoke isr() here!
        *y = t;
        t = s;  // restore global variable
}

void isr()
{
        int x = 1, y = 2;
        swap(&x, &y);
}
```

Exemplu

```
[ServiceContract]
public interface IStockService
{ // metode
}

[ServiceBehavior(ConcurrencyMode = ConcurrencyMode.Single)]
class StockService : IStockService
{ // cod }
```

In continuare se prezinta un tabel cu relatia dintre **InstanceContextMode** si **ConcurrencyMode**.

	InstanceContextMode Single	**InstanceContextMode PerCall**	**InstanceContextMode PerSession**
ConcurrencyMode Single *(Numai un fir la un moment dat poate accesa clasa serviciului)*	**Singleton** : O instanta si un fir. Cererile sunt puse intr-o coada FIFO.	O instanta este creata pentru fiecare apel. Fiecare instanta are firul sau de executie.	O instanta este creata pentru fiecare sesiune a clientului si numai un fir este creat pentru a procesa cererea pentru aceasta sesiune. Daca clientul face cereri multiple pe aceeasi sesiune se foloseste o coada FIFO pentru a asigura ordinea de procesare.
ConcurrencyMode Reentrant *(Numai un fir la un moment dat poate accesa clasa serviciului, dar firul poate parasi clasa si poate reveni mai tarziu pentru a-si continua executia)*	**Singleton** : O instanta si un fir. Cererile sunt puse intr-o coada FIFO. Firul poate parasi metoda, sa lucreze pe alt fir, apel asincron, callback din alt serviciu si sa se reintoarca mai tarziu.	O instanta este creata pentru fiecare apel. Fiecare instanta are firul sau de executie.	O instanta este creata pentru fiecare sesiune a clientului si numai un fir este creat pentru a procesa cererea pentru aceasta sesiune. Daca clientul face cereri multiple pe aceeasi sesiune se foloseste o coada FIFO pentru a asigura ordinea de procesare. Firul poate parasi metoda, sa execute altceva si sa se reintoarca mai traziu.
ConcurrencyMode Multiple *(Fire multiple pot accesa clasa serviciului in mod simultan)*	O instanta este creata dar fire multiple pot lucra in paralel pe aceasta instanta. Membrii calsei trebuie protejati prin sincronizare.	O instanta este creata pentru fiecare apel. Fiecare instanta are firul sau de executie.	O instanta este creata pentru fiecare sesiune a clientului, dar fire multiple pot rula in paralel pe aceasta instanta. Apelurile multiple de la client se proceseaza in paralel. Datele membru din clasa trebuie sa fie protejate prin sincronizare.

Cateva scenarii legate de instanta serviciului, binding si concurenta

1. Concurenta implicita si intantiere cu binding fara sesiune

In acest caz se va crea o noua instanta a serviciului pentru fiecare cerere pe care o primeste si executa codul pe firul propriu.

Fiind binding fara sesiune, **`InstanceContextMode.PerSession`** va fi de fapt **`InstanceContextMode.PerCall`**.

Exemplu

```
[ServiceContract]
public interface IStockService
{ // metode
}
[ServiceBehavior(ConcurrencyMode = ConcurrencyMode.Single,
     InstanceContextMode = InstanceContextMode.PerSession)]
class StockService : IStockService
{ // cod }
```

2. O singura instanta si fire multiple

Pentru a crea o singura instanta a serviciului ce este partajata de fire concurente trebuie sa folosim **`InstanceContextMode.Single`** si **`ConcurrencyMode.Multiple`**.

Trebuie sa facem sincronizare pentru ca firul sa-si protejeze memorarile locale.

```
[ServiceContract]
public interface IStockService
{ // metode
}

[ServiceBehavior(ConcurrencyMode = ConcurrencyMode.Multiple,
     InstanceContextMode = InstanceContextMode.Single)]
class StockService : IStockService
{ // cod }
```

3. Implementare Singleton

```
[ServiceContract]
public interface IStockService
{ // metode
}

[ServiceBehavior(InstanceContextMode = InstanceContextMode.Single)]
class StockService : IStockService
```

```
{ // cod }
```

4. Instante la nivel de sesiune

Pe site-urile Web sau in aplicatii Web se foloseste memorarea starii sesiunii. In acest caz exista o corespondenta intre utilizatori si sesiune. WCF suporta un concept similar in ceea ce priveste serviciile.

Pentru a implementa instante ale serviciului *per-session*, trebuie sa permitem sesiuni la nivel de contract si la nivel de serviciu.

La nivel de *contract* se foloseste comportarea **`SessionMode`**.

La nivel de *serviciu* se foloseste **`InstanceContextMode.PerSession`**.

`SessionMode` specifica valorile disponibile pentru a indica suportul pentru sesiuni de incredere la nivel de *contract*.

Valori posibile :

- **`Allowed`** Contractul suporta sesiuni daca binding-ul pentru primire suporta de asemenea sesiuni de incredere.
- **`Required`** Contractul are nevoie de un binding ce suporta sesiunea. Se genereaza o exceptie daca binding-ul nu suporta sesiune.
- **`NotAllowed`** Contractul nu suporta binding-uri ce initiaza sesiunea.

O sesiune este o modalitate de a corela o multime de mesaje schimbate intre doua sau mai multe endpoint-uri.

Sesiunea este un concept la nivel de canal.

Din aceasta cauza exista o interdependenta intre **`SessionMode`** si **`InstanceContextMode`**.

MSDN

Canale cu sesiune (coloana din stanga) si fara sesiune (coloana din dreapta).

Instance Context Mode	SessionMode.Required Cu sesiune | Fara sesiune		SessionMode.Allowed Cu sesiune |Fara sesiune		SessionMode. NotAllowed	
PerCall	O sesiune si o instanta pt fiecare apel.	Se genereaza o exceptie	O sesiune si o instanta pt fiecare apel	O instanta pentru fiecare apel	Se genereaza o exceptie	O instanta pt fiecare apel
PerSession	O sesiune si o instanta pentru fiecare canal.	Se genereaza o exceptie	O sesiune si o instanta pt fiecare canal.	O instanta pentru fiecare apel.	Se genereaza o exceptie.	O instanta pt fiecare apel.
Single	O sesiune si o instanta pentru toate apelurile.	Se genereaza o exceptie.	O sesiune si o instanta pt fiecare singleton creat sau pt singleton specificat de utilizator.	O instanta pt fiecare singleton creat sau pt singleton specificat de utilizator.	Se genereaza o exceptie.	O instanta pentru fiecare singleton creat sau pentru singleton specificat de utilizator.

Desi sesiunile sunt specificate la nivel de contract, ele sunt implementate in canalul specificat de elementele *binding*. Deci, aceasta comportare a contractului verifica daca contractul si canalul sunt compatibile cand se lanseaza serviciul. De exemplu, daca canalul cere sesiune, dar binding-ul utilizat nu suporta sesiune (**httpBinding**), atunci cerintele pentru sesiune ale contractului nu pot fi indeplinite si se genereaza o exceptie cand se lanseaza serviciul.

```
[ServiceContract(SessionMode=SessionMOde.Required)]
public interface IStockService
{ // metode
}

[ServiceBehavior(ConcurrencyMode = ConcurrencyMode.Multiple,
     InstanceContextMode = InstanceContextMode.PerSession)]
class StockService : IStockService
{ // cod }
```

Controlarea numarului de instante concurente

```
<serviceThrottling maxConcurentInstances="10">
```

Implicit, WCF va crea atatea instante de cate sunt nevoie si va aloca fire necesare sa raspunda cerintelor. In teorie totul e OK, dar in practica ...

WCF are posibilitatea de a limita numarul maxim de instante pe care le poate crea. Acest lucru se face folosind elementul **serviceThrottling** din fisierul de configurare, sectiunea **behavior**.

```
<behaviors>
   <serviceBehaviors>
        <behavior name= "concurenta">
             <serviceThrottling maxConcurentInstances="10">
        </behavior>
   </serviceBehaviors>
</behaviors>
```

Observatie
Se foloseste in special in cazurile cand se genereaza o noua instanta a serviciului la fiecare apel.

Controlarea numarului de apeluri concurente

```
<serviceThrottling maxConcurentCalls="10">
```

Cand modul de instantiere este specificat ca **Single**, WCF creaza o singura instanta in host, indiferent de numarul de clienti ce au facut cereri.

Cand **ConcurrencyMode** este specificat ca **Multiple,** WCF creaza un fir pentru fiecare cerere (pana la limita data de sistem), pentru executia paralela a metodelor din serviciu.

Pentru a gestiona acest lucru, se foloseste **maxConcurrentCalls** (se gaseste sub elementul **serviceThrottling**) ce specifica numarul maxim de fire concurente gestionate de serviciu pentru a raspunde cererilor clientilor.

Exemplu

```
[ServiceContract]
public interface IStockService
{ // metode
}

[ServiceBehavior(ConcurrencyMode = ConcurrencyMode.Multiple,
     InstanceContextMode = InstanceContextMode.Single)]
class StockService : IStockService
{ // cod }
```

iar in fisierul de configurare vom avea:

```
<behaviors>
   <serviceBehaviors>
```

```
        <behavior name= "concurenta">
             <serviceThrottling maxConcurentCalls="10">
        </behavior>
   </serviceBehaviors>
</behaviors>
```

Controlarea numarului sesiunilor concurente

```
<serviceThrottling maxConcurentSessions="10">
```

Cand este definit **InstanceContextMode.PerSession**, WCF creaza o instanta pentru fiecare sesiune ce se conecteaza la serviciu.

Pentru a controla numarul de sesiuni conectate la serviciu, folosim **maxConcurrentSession** care se gaseste sub elementul **serviceThrottling.**

Cand numarul maxim de sesiuni concurente este atins, urmatorul client ce incearca sa creeze o sesiune va astepta pana cand o alta sesiune este inchisa. Aceasta setare este folositoare atunci cand dorim sa limitam numarul utilizatorilor (clienti sau servere) ce se pot conecta la un serviciu.

Exemplu

```
[ServiceContract(SessionMode=SessionMOde.Required)]
public interface IStockService
{ // metode
}

[ServiceBehavior(ConcurrencyMode = ConcurrencyMode.Multiple,
     InstanceContextMode = InstanceContextMode.PerSession)]
class StockService : IStockService
{ // cod }
```

iar fisierul de configurare vom avea ceva de genul :

```
...
<behaviors>
   <serviceBehaviors>
        <behavior name= "concurenta">
             <serviceThrottling maxConcurentSessions="10">
        </behavior>
   </serviceBehaviors>
</behaviors>
...
```

Implementarea tranzactiilor (Comportare operatie)

Referitor la tranzactii, exista doua scenarii :

- **procese de business in mai multi pasi** – necesita minute sau zile pentru a se finaliza, au multe dependente externe.
- **tranzactii scurte** – operatii de business de ordinul secundelor (ca timp de executie) si care au putine dependente externe.

Pentru primul scenariu (nu il discutam, combina workflow manual si/sau automatizat) se aplica un « broker » de mesaje sau un « bus » de servicii, de ex. BizTalk Sever, Message Broker de la IBM, etc..

WCF suporta scenariul 2 - **tranzactii scurte**.

Operatii tranzactionale in interiorul unui serviciu

Operatiile tranzactionale in interiorul unui serviciu se executa ca o unitate, ori toate se executa complet ori toate esueaza. Acestea sunt initiate de o singura parte a serviciului. Clientul nu are nimic de a face cu semantica tranzactionala, el apeleaza doar o metoda pe serviciu. Totul se executa pe partea de serviciu.

Pentru a implementa o asemenea comportare in WCF, operatia pe serviciu trebuie sa fie marcata ca fiind tranzactionala (sa suporte tranzactii) folosind

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

Putem indica faptul ca o operatie este completa in mod ***implicit*** sau ***explicit***.
Modul *implicit* presupune folosirea atributelor, ca in exemplul de mai jos:

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

In acest caz operatia se considera completa daca nu genereaza o eroare.

Modul ***explicit*** presupune scrierea de cod in metodele din serviciu.

Se foloseste atributul

```
[OperationBehavior(TransactionAutoComplete=false)]
```

si se apeleaza in mod explicit

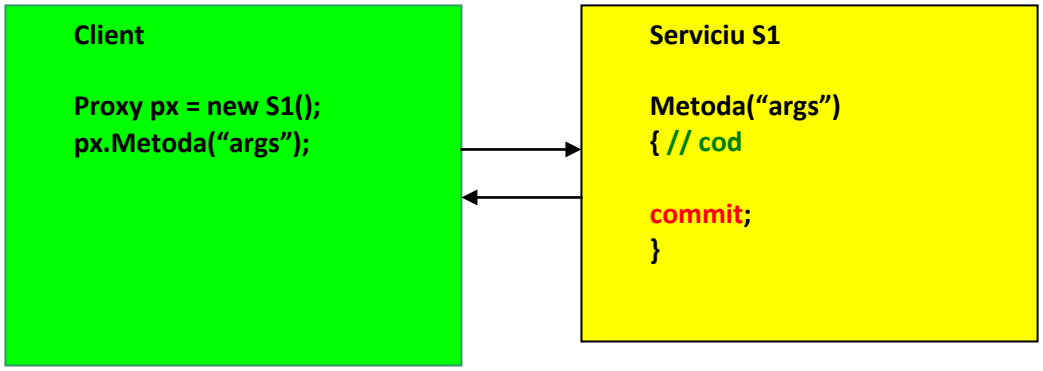
```
OperationContext.Current.SetTransactionComplete();
```

inainte de a returna din metoda.

Daca folosim metoda explicita, trebuie de asemenea sa folosim un binding bazat pe sesiune in canalul de comunicatie si de asemenea trebuie sa suportam sesiuni in contractul serviciului folosind atributul :

```
[ServiceContract(SessionMode=SessionMode.Allowed)]
```

O schema de tranzactie simpla.



Se observa ca metoda din serviciu executa commit.

Promovarea tranzactiilor intre operatii (*transaction flow*)

Cand se lucreaza cu sisteme distribuite, tranzactiile uneori trebuie sa depaseasca frontierele serviciului. De exemplu, daca un serviciu gestioneaza informatia despre clienti si un alt serviciu gestioneaza comenzile, si un utilizator doreste sa plaseze o comanda si sa vanda produsul la o alta adresa, sistemul ar trebui sa aiba nevoie sa invoce operatii pe fiecare serviciu. Daca tranzactia s-a terminat, utilizatorul se asteapta ca ambele sisteme sa faca actualizarile separat si in mod corect. Daca infrastructura suporta protocolul tranzactional atomic (vezi proprietatile ACID), serviciile pot fi compuse intr-o tranzactie agregat.

WS-AT (Web Srvice Atomic Transaction) furnizeaza infrastructura pentru a partaja informatia intre serviciile participante, pentru a implementa semantica ***commit*** in doua etape *(two-phase commit*) necesara pentru tranzactiile ACID *(atomicity, consistency, isolation, durability).*

Memento ACID
A : « totul sau nimic » ;
C : « baza de date trece dintr-o stare consistenta in alta stare consistenta » ;
I : « o tranzactie nu interfera cu o alta tranzactie » ;
D : « modificarea este realizata si orice cadere de sistem nu o poate modifica ».

In WCF, promovarea informatiei tranzactionale peste frontierele serviciului se numeste **« *transaction flow* ».**

Pentru a promova o tranzactie peste granitele unui serviciu, trebuie executati urmatorii pasi:

- **`[ServiceContract(SessionMode=SessionMode.Required)]`**
  - Serviciul are nevoie de sesiune pentru ca aceasta informatie va fi partajata intre coordonator (in mod obisnuit clientul) si serviciile participante.

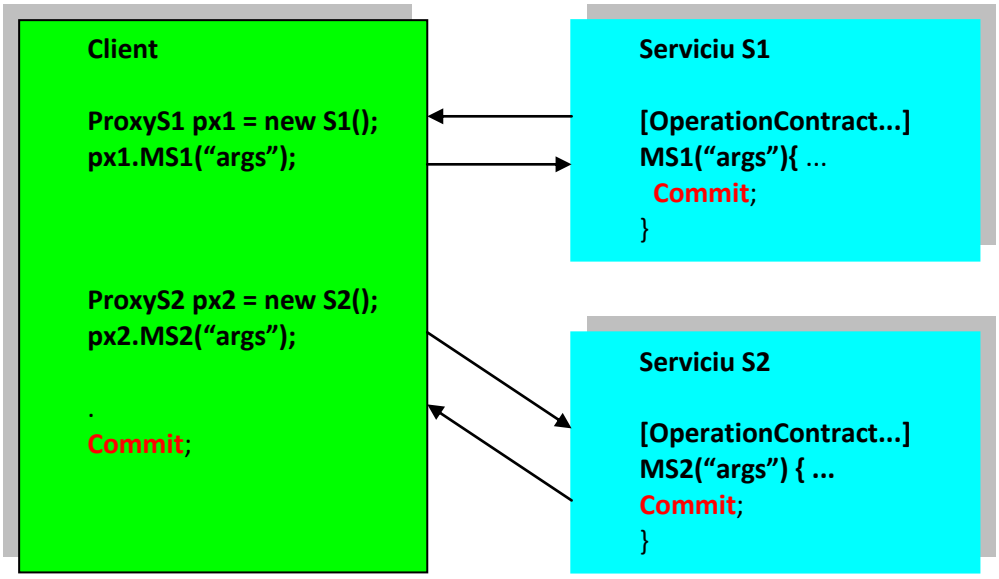
- **`[OperationBehavior(TransactionScopeRequired=true)].`**
  - Operatia trebuie sa ceara o tranzactie.
  - Aceasta va crea o noua tranzactie daca nu e deja creata.

- **[OperationContract(TransactionFlowOption = TransactionFlowOption.Allowed)].**

- Operatia trebuie sa permita ca informatia despre tranzactie sa fie plasata in antetul (header) mesajului.

➢ **(Binding definition) TransactionFlow=true**.
- Binding-ul trebuie sa permita tranzactia astfel incat canalul sa poata sa puna informatia despre tranzactie in header SOAP.
- Binding-ul trebuie sa suporte sesiuni (**`wsHttpBinding`** suporta sesiuni, **`httpBinding`** nu suporta sesiuni).

➢ **(Client) TransactionScope.**
- Partea ce initiaza tranzactia, in mod normal clientul, trebuie sa foloseasca un domeniu al tranzactiei ("*transaction scope*") cand apeleaza operatiile serviciului.
- Acesta trebuie sa apeleze **TransactionScope.Close()** pentru ca modificarile sa devina efective (commit).



Proprietatea **`OperationBehaviorAttribute.TransactionScopeRequired`**

este Read/Write iar valoarea proprietatii indica daca metoda are cerinte de tranzactii pentru a se executa.

Daca o tranzactie este disponibila, operatia se executa in acea tranzactie, in caz contrar se creaza o noua tranzactie ce va fi folosita pentru executarea operatiei. Binding-ul specificat in endpoint controleaza daca sunt suportate astfel de tranzactii.

Legatura dintre binding si tranzactii este data in urmatorul tabel (MSDN).

TransactionScopeRequired	Binding permits transaction flow	Caller flows transaction	Result
False	False	No	Method executes without a transaction.
True	False	No	Method creates and executes within a new transaction.
True or False	False	Yes	A SOAP fault is returned for the transaction header.
False	True	Yes	Method executes without a transaction.
True	True	Yes	Method executes under the flowed transaction.

TransactionFlowOption

MSDN

Member name	Description
NotAllowed	A transaction should not be flowed. This is the default value.
Allowed	Transaction **may** be flowed.
Mandatory	Transaction **must** be flowed.

Comportarea tranzactionala se defineste in clasa **`TransactionScope`** folosind in ctor acestei clase o valoare data de **`TransactionScopeOption`**.

Valorile posibile pentru **`TransactionScopeOption`** sunt cele din tabelul de mai jos (MSDN).

TransactionScopeOption

Member name	Description
Required	A transaction is required by the scope. It uses an ambient transaction if one already exists. Otherwise, it creates a new transaction before entering the scope. This is the default value.
RequiresNew	A new transaction is always created for the scope.

Suppress	The ambient transaction context is suppressed when creating the scope. All operations within the scope are done without an ambient transaction context.

TransactionScopeOption este folosita in ctor clasei **TransactionScope** pentru a defini comportarea tranzactionala a domeniului.

Clasa TransactionScope

O activitate (workflow) ce demarcheaza frontierele unei tranzactii.

Face un bloc de cod tranzactional.

Infrastructura **`System.Transaction`** furnizeaza un model *explicit* bazat pe clasa **`Transaction`** si un model *implicit* bazat pe clasa **`TransactionScope`**, in care tranzactiile sunt gestionate in mod automat de infrastructura.

Cand aceasta activitate isi incepe executia, o noua tranzactie este creata daca nu exista deja una. Tranzactia face « commit » cand activitatea si toti ceilalti participanti in tranzactie s-au executat cu succes.

In MSDN TransactionScope este descrisa astfel:

Upon instantiating a TransactionScope by the **new** statement, the transaction manager determines which transaction to participate in. Once determined, the scope always participates in that transaction. The decision is based on two factors: whether an ambient transaction is present and the value of the **TransactionScopeOption** parameter in the constructor. The ambient transaction is the transaction your code executes in. You can obtain a reference to the ambient transaction by calling the static Current property of the Transaction class. For more information on how this parameter is used, please see the "Transaction Flow Management" section of the Implementing An Implicit Transaction Using Transaction Scope topic.

If no exception occurs within the transaction scope (that is, between the initialization of the TransactionScope object and the calling of its Dispose method), then the transaction in which the scope participates is allowed to proceed. If an exception does occur within the transaction scope, the transaction in which it participates will be rolled back.

When your application completes all work it wants to perform in a transaction, you should call the Complete method only once to inform that transaction manager that it is acceptable to commit the transaction. Failing to call this method aborts the transaction.

A call to the Dispose method marks the end of the transaction scope. Exceptions that occur after calling this method may not affect the transaction.

If you modify the value of Current inside a scope, an exception is thrown when Dispose is called. However, at the end of the scope, the previous value is restored. In addition, if you call Dispose on Current inside a transaction scope that created the transaction, the transaction aborts at the end of the scope.

Exemplu (MSDN)

```
// This function takes arguments for 2 connection strings and
// commands to create a transaction involving two SQL Servers.
// It returns a value > 0 if the transaction is committed, 0 if the
// transaction is rolled back. To test this code, you can connect
// to two different databases on the same server by altering the
// connection string, or to another 3rd party RDBMS by
// altering the code in the connection2 code block.
static public int CreateTransactionScope(
    string connectString1, string connectString2,
    string commandText1, string commandText2)
{
    // Initialize the return value to zero and create a
    // StringWriter to display results.
    int returnValue = 0;
    System.IO.StringWriter writer = new System.IO.StringWriter();

    try
    {
        // Create the TransactionScope to execute the commands, guaranteeing
        // that both commands can commit or roll back as a single
        // unit of work.
        using (TransactionScope scope = new TransactionScope())
        {
            using (SqlConnection connection1 =
                    new SqlConnection(connectString1))
            {
                // Opening the connection automatically enlists it in the
                // TransactionScope as a lightweight transaction.
                connection1.Open();

                // Create the SqlCommand object and
             // execute the first command.
                SqlCommand command1 =
                    new SqlCommand(commandText1, connection1);
                returnValue = command1.ExecuteNonQuery();
                writer.WriteLine("Rows to be affected by command1: {0}",
                            returnValue);

                // If you get here, this means that command1 succeeded.
             // By nesting the using block for connection2 inside
             // that of connection1, you conserve server and network
             // resources as connection2 is opened
                // only when there is a chance that the transaction can
             // commit.
                using (SqlConnection connection2 =
                    new SqlConnection(connectString2))
                {
                    // The transaction is escalated to a full distributed
                    // transaction when connection2 is opened.
                    connection2.Open();

                    // Execute the second command in the second database.
                    returnValue = 0;
                    SqlCommand command2 =
                            new SqlCommand(commandText2, connection2);
                    returnValue = command2.ExecuteNonQuery();
                    writer.WriteLine("Rows to be affected by command2: {0}",
                            returnValue);
```

```
                    }
                }
            }

            // The Complete method commits the transaction.
        // If an exception has been thrown,
            // Complete is not  called and the transaction is rolled back.
            scope.Complete();
        }
        catch (TransactionAbortedException ex)
        {
            writer.WriteLine("TransactionAbortedException Message: {0}",
                  ex.Message);
        }
        catch (ApplicationException ex)
        {
            writer.WriteLine("ApplicationException Message: {0}", ex.Message);
        }

        // Display messages.
        Console.WriteLine(writer.ToString());

        return returnValue;
    }
```

Clasa Transaction

Reprezinta o tranzactie.

Spatiul de nume **System.Transactions** furnizeaza un model de programare ***explicit*** bazat pe clasa **`Transaction`** si un model de programare ***implicit*** bazat pe clasa **`TransactionScope`**.

Membri

Name	Description
EnlistDurable(Guid, IEnlistmentNotification, EnlistmentOptions)	Enlists a durable resource manager that supports two phase commit to participate in a transaction.
EnlistDurable(Guid, ISinglePhaseNotification, EnlistmentOptions)	Enlists a durable resource manager that supports single phase commit optimization to participate in a transaction.
EnlistPromotableSinglePhase	Enlists a resource manager that has an internal transaction using a promotable single phase enlistment (PSPE).
EnlistVolatile(IEnlistmentNotification, EnlistmentOptions)	Enlists a volatile resource manager that supports two phase commit to participate in a transaction.
EnlistVolatile(ISinglePhaseNotificati	Enlists a volatile resource manager that

on, EnlistmentOptions)	supports single phase commit optimization to participate in a transaction.
Rollback()	Rolls back (aborts) the transaction.
Rollback(Exception)	Rolls back (aborts) the transaction.

Contracte de date

In interiorul unui serviciu functionalitatea aplicatiei este implementata in *cod*.

In afara serviciului functionalitatea este definita in *WSDL*.

In interiorul unui serviciu WCF, data este reprezentata de tipuri simple sau complexe in timp ce in exteriorul lui data este reprezentata de XML Schema Definition (XSD).

Contractul de date din WCF furnizeaza o mapare intre tipurile din .NET CLR definite in cod si XSD, [specificatie definita de organizatia W3C (www.w3c.org)].

Tipurile din XSD sunt utilizate pentru comunicatie in afara serviciului.

In momentul proiectarii unui tip ce va fi transferat in si de la client, folosim atributul [**DataContract**] pentru a indica ce clase trebuie sa fie reprezentate ca XSD si incluse in WSDL expus de serviciu.

Atributul [**DataMemmber**] defineste membrii din clasa ce vor fi inclusi in reprezentarea externa.

La runtime clasa [**DataContractSerializer**] serializeaza obiectele in XML folosind reguli descrise de atributele [**DataContract**] si [**DataMember**].

DataContractSerializer va serializa tipurile si le va expune in contractele WSDL daca acestea indeplinesc urmatoarele conditii :

- Tipuri marcate cu atributele [**DataContract**] si [**DataMember**] ;
- Tipuri marcate cu atributul [**CollectionDataContract**] ;
- Tipuri derivate din **IXmlSerializable** ;
- Tipuri marcate cu atributul [**Serializable**] si ai carui membri nu sunt marcati cu [**NonSerialized**].
- Tipuri marcate cu atributul [**Serializable**] si implementeaza **ISerializable**.
- Tipuri primitive preconstruite in CLR (int32, string).
- Array de bytes, **DateTime**, **TimeSpan**, **Guid**, **Uri**, **XmlQualifiedName**, **XmlElement** si **XmlNode** ;
- Arrays si colectii cum ar fi :**List<T>, Dictionary<K,U>, Hashtable.**
- Enumerari**.**

Domeniul membrilor clasei .NET, indiferent ca este *public* sau *private*, nu are impact asupra includerii acestuia in schema XML.

Atributul **[DataMember]** defineste ca data sa fie inclusa in schema XML.

Exemplu

```
[DataContract]
class Customer
{
   [DataMember(Name=”CustomerName”, Order=0, IsRequired=true)]
```

```
    private string Nume;

    [DataMember(Name="Id", Order=1, IsRequired=true)]
    private int IdCustomer;
}
```

Observatie
svcutil.exe cu switch **–t:metadata** genereaza XSD.

Serializare

Contractul de date este parte a operatiilor contractuale pe care le suporta serviciul, la fel cum contractul serviciului este parte a contractului.

Contractul de date este publicat in metadata si permite clientilor de a converti reprezentarea neutra a datelor intr-o reprezentare nativa (tipuri CLR .NET).

Obiectele locale si referintele la acestea constituie concepte ale CLR.

Nu se poate transmite cod sau parti logice ale invocarilor de metode de pe partea de client catre serviciu. Ceea ce se transmite este starea unui obiect si pornind de aici serviciul si/sau clientul vor reconstrui obiectul.

Starea obiectului se transmite prin ceea ce se numeste *serializare* si se construieste in timpul operatiei de *deserializare*.

.NET serializeaza si deserializeaza obiectele folosind reflection (instrospectia metadatei).
.NET preia valorile fiecarui camp al unui obiect si le serializeaza in memorie sau in fisier sau peste o conexiune in retea. Pentru deserializare, .NET creaza un nou obiect de tipul dat, citeste valorile persistente si seteaza campurile cu aceste valori, folosind din nou refelection. (Reflection poate accesa campurile private cat si cele din clasa de baza.)

Un stream este o secventa logica de « bytes », si nu depinde de mediul particular cum ar fi fisier, memorie, port de comunicatie sau alta resursa.

Tipurile construite de utilizator (clase si structuri) nu sunt serializabile in mod implicit. Dezvoltatorul tipului decide daca acesta poate fi serializat sau nu.
Pentru a indica ca un tip este serializabil se poate adnota tipul cu atributul **`SerializableAttribute`** definit astfel :

```
[AttributeUsage(AttributeTargets.Delegate|
                AttributeTargets.Enum    |
                AttributeTargets.Struct  |
                AttributeTargets.Class,
                Inherited=false)]
public sealed class SerializableAttribute : Attribute
{}
```

Exemplu

```
[Serializable]
public class MyClass
{...}
```

Adnotand astfel un tip, inseamna ca *toti* membrii tipului trebuie sa fie serializabili.

Daca dorim ca un anumit membru sa nu fie serializabil (chiar daca poate fi serializat) adnotam acel membru cu atributul [**NonSerialized**].

```
public class MyOtherClass
{..}

[Serializable]
public class MyClass
{
   [NonSerialized]
   MyOtherClass m_OtherClass;
   /* Methods and properties */

   [NonSerialized]
   string strInfo;          // desi string poate fi serializat in .NET
}
```

Formate .NET pentru serializare

Exista doua formate:

- **BinaryFormatter** – serializare intr-o forma binara compacta, serializare rapida.
- **SoapFormatter** – foloseste formatul XML SOAP specific pentru .NET.

Ambele formate suporta interfata **IFormatter**, definita astfel:

```
public interface IFormatter
{
   object Deserialize(Stream serializationStream);
   void Serialize(Stream serializationStream,object graph);
   // More members
}

public sealed class BinaryFormatter : IFormatter,...
{...}
public sealed class SoapFormatter : IFormatter,...
{...}
```

Interfata **IFormatter** defineste metodele **Serialize** si **Deserialize**, implementate de clasele **BinaryFormatter** si **SoapFormatter**.

Indiferent de formatul folosit, ambele formate salveaza in stream si informatii despre versionare si assembly-ul unde sunt definite tipurile. Din cauza ca se utilizeaza informatii despre assembly, acest tip de serializare nu poate fi folosit in crearea serviciilor (clientii si serverul trebuie sa aiba acelasi assembly)

.

Folosirea unui **Stream** este iarasi o problema in acest scenariu deorece serverul si clientii trebuie sa partajeze stream-ul.

Formate WCF pentru serializare

Clasa folosita in acest caz este **DataContractSerializer**.

DataContractSerializer : definit in **System.Runtime.Serialization** si derivat din **XmlObjectSerializer**.

```
public abstract class XmlObjectSerializer
{
   public virtual object ReadObject(Stream stream);
   public virtual object ReadObject(XmlReader reader);
   public virtual void WriteObject(XmlWriter writer,object graph);
   public void WriteObject(Stream stream,object graph);
   //More members
}

public sealed class DataContractSerializer : XmlObjectSerializer
{
   public DataContractSerializer(Type type);
   //More members
}
```

Metodele pentru serializare si deserializare sunt **ReadObject**, respectiv **WriteObject**.

DataContractSerializer *nu* suporta interfata **IFormatter**.

WCF foloseste in mod automat **DataContractSerializer**, iar ca dezvoltatori nu trebuie sa scriem cod.

Observatie
Se poate folosi **DataContractSerializer** pentru a serializa tipuri in si din stream .NET, iar in ctor va trebui sa furnizam tipul pe care dorim sa-l folosim.

```
MyClass obj1 = new MyClass( );
DataContractSerializer formatter =
            new DataContractSerializer(typeof(MyClass));

using(Stream stream = new MemoryStream())
{
   // memorare (salvare) obiect
   formatter.WriteObject(stream,obj1);
   // refacere obiect
   stream.Seek(0,SeekOrigin.Begin);
   MyClass obj2 = (MyClass)formatter.ReadObject(stream);
}
```

Exemplu cu DataContractSerializer

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Text;
using System.Xml;
using System.ServiceModel;

namespace DataContractSerialization
{
    class Program
    {
        static void Main(string[] args)
        {
            Student student = new Student()
            {
                Age = 12,
                Name = "David"
            };

            DataContractSerializer dcs =
               new DataContractSerializer(typeof(Student));
            StringBuilder sb = new StringBuilder();

            /*
            public static XmlWriter Create(StringBuilder output)

            Parameters
            output
            Type: System.Text.StringBuilder
            The StringBuilder to which to write to. Content written
               by the XmlWriter is appended to the StringBuilder.

            Return Value
            Type: System.Xml.XmlWriter
            An XmlWriter object.

            */

            using (XmlWriter writer = XmlWriter.Create(sb))
            {
                dcs.WriteObject(writer, student);
            }
            string xml = sb.ToString();

            Console.WriteLine(xml);

            using (XmlTextReader reader = new XmlTextReader(
                      new StringReader(xml)))
            {
                Student newStudent = (Student)dcs.ReadObject(
                                                    reader, true);


               Console.WriteLine(
               "Obiectul construit dupa deserializare");
                Console.WriteLine("newStudent : Name = {0}, Age = {1}",
               newStudent.Name, newStudent.Age);
```

```
            }

            Console.ReadKey();
        }
    }

    [DataContract(Namespace = "http://www.netspecial.com")]
    public class Student
    {
        [DataMember]
        public string Name { get; set; }

        [DataMember]
        public int Age { get; set; }
    }
}
```

```
Rezultatul este :
<?xml version="1.0" encoding="utf-16"?>
<Student xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://www.netspecial.com">
   <Age>12</Age>
   <Name>David</Name>
</Student>
Obiectul construit dupa deserializare
newStudent : Name = David, Age = 12
```

Cazul cand se foloseste un stream (MSDN)

```
using System;
using System.IO;
using System.Xml;
using System.Text;

public class Sample
{

    public static void Main()
    {

        XmlWriter writer = null;

        try
        {
            // Create an XmlWriterSettings object with
         // the correct options.

            XmlWriterSettings settings = new XmlWriterSettings();
            settings.Indent = true;
            settings.IndentChars = ("\t");
            settings.OmitXmlDeclaration = true;

            // Create the XmlWriter object and write some content.

            writer = XmlWriter.Create("data.xml", settings);
            writer.WriteStartElement("book");
            writer.WriteElementString("item", "tesing");
            writer.WriteEndElement();
```

```
            writer.Flush();
        }
        finally
        {
            if (writer != null)
                writer.Close();
        }
    }
}
```

iar fisierul data.xml va contine:

```
<book>
   <item>tesing</item>
</book>
```

Contract de date via serializare

Toate tipurile primitive construite in .NET sunt serializabile.

Pentru a putea folosi un tip construit de utilizator ca un parametru al unei operatii trebuiesc indeplinite urmatoarele conditii:

- Tipul trebuie sa fie serializabil.
- Clientul si serverul trebuie sa aiba definitia locala a tipului.

Exemplu

```
[Serializable]
struct Contact
{
   public string FirstName;
   public string LastName;
}

[ServiceContract]
interface IContactManager
{
   [OperationContract]
   void AddContact(Contact contact);

   [OperationContract]
   Contact[] GetContacts( );
}
```

Clientul poate utiliza o definitie echivalenta a tipului, ca in exemplul urmator:

```
[Serializable]
struct Contact
{
   public string FirstName;
   public string LastName;

   [NonSerialized]
```

```
    public string Address;
}
```

Observati data membru *Address* care nu este definita in contractul de baza.

Atributul `DataContract`

`DataContractAttribute` este definit in spatiul de nume **`System.Runtime.Serialization`** astfel:

```
[AttributeUsage(AttributeTargets.Enum  |
                AttributeTargets.Struct|
                AttributeTargets.Class,
                Inherited = false)]
public sealed class DataContractAttribute : Attribute
{
   public string Name
   {get;set;}
   public string Namespace
   {get;set;}
}
```

Aplicand acest atribut unei clase sau structuri nu este suficient pentru ca WCF sa serializeze membrii sai.

```
[DataContract]
struct Contact
{
   //Nu vor fi parte a contractului de date
   public string FirstName;
   public string LastName;
}
```

Fiecare camp va fi membru al contractului de date daca este adnotat cu atributul **`DataMemberAttribute`**, definit astfel :

```
[AttributeUsage(AttributeTargets.Field|AttributeTargets.Property,
                Inherited = false)]
public sealed class DataMemberAttribute : Attribute
{
   public bool IsRequired
   {get;set;}
   public string Name
   {get;set;}
   public int Order
   {get;set;}
}
```

Acest atribut se poate aplica pe *campuri*:

```
[DataContract]
struct Contact
{
```

```
    [DataMember]
    public string FirstName;

    [DataMember]
    public string LastName;
}
```

sau *proprietati*:

```
[DataContract]
struct Contact
{
    string m_FirstName;
    string m_LastName;

    [DataMember]
    public string FirstName
    {
        get
        {...}
        set
        {...}
     }

    [DataMember]
    public string LastName
    {
        get
        {...}
        set
        {...}
     }
}
```

Nu e necesar ca modificatorul de acces sa fie public.

```
[DataContract]
struct Contact
{
    [DataMember]
    string m_FirstName;

    [DataMember]
    string m_LastName;
}
```

Contractele de date sunt case sensitive, la nivel de tip si membru.

Import contract de date

Cand un contract de date este folosit intr-o operatie a contractului, acesta este publicat in metadata. Cand clientul importa definitia contractului de date, acesta va primi o definitie *echivalenta* cu cea din server, dar nu identica.

Se mentine tipul original al clasei sau structurii si tipul spatiului de nume.

De exemplu, pentru:

```
namespace MyNamespace
{
    [DataContract]
    struct Contact
    {...}

    [ServiceContract]
    interface IContactManager
    {
        [OperationContract]
        void AddContact(Contact contact);

        [OperationContract]
        Contact[] GetContacts( );
    }
}
```

definitia importata va fi:

```
namespace MyNamespace
{
    [DataContract]
    struct Contact
    {...}
}
[ServiceContract]
interface IContactManager
{
    [OperationContract]
    void AddContact(Contact contact);

    [OperationContract]
    Contact[] GetContacts( );
}
```

Se poate furniza un alt spatiu de nume prin atribuirea unei valori proprietatii **Namespace** din atributul **DataContract**.

```
namespace MyNamespace
{
    [DataContract(Namespace = "MyOtherNamespace")]
    struct Contact
```

```
    {...}
}
```

Definitia importata va fi:

```
namespace MyOtherNamespace
{
    [DataContract]
    struct Contact
    {...}
}
```

Definitia importata va avea totdeauna *proprietatile* adnotate cu atributul **DataMember**, chiar daca tipul original de pe partea de serviciu nu defineste proprietati.

Campurile importate vor fi sufixate cu cuvantul **Field**.

```
[DataContract]
struct Contact
{
    [DataMember]
    public string FirstName;

    [DataMember]
    public string LastName;
}
```

iar ceea ce se importa in client va fi:

```
[DataContract]
public partial struct Contact
{
    string FirstNameField;
    string LastNameField;

    [DataMember]
    public string FirstName
    {
        get
        {
            return FirstNameField;
        }
        set
        {
            FirstNameField = value;
        }
    }

    [DataMember]
    public string LastName
    {
        get
        {
            return LastNameField;
        }
        set
        {
            LastNameField = value;
```

```
        }
    }
}
```

Observatie
Se poate modifica aceasta definitie pe partea de client (manual).

Modificatorul de acces al campurilor importate va fi totdeauna **public**, indiferent de cum sunt definite pe partea de server.

```
[DataContract]
struct Contact
{
    [DataMember]
    string FirstName
    {get;set;}

    [DataMember]
    string LastName;
}
```

Daca atributul **DataMember** este aplicat unei *proprietati* pe partea de server, definitia importata va fi tot o *proprietate* si va avea numele proprietatii din server.

```
[DataContract]
public partial struct Contact
{
    string m_FirstName;
    string m_LastName;

    [DataMember]
    public string FirstName
    {
        get
        {
            return m_FirstName;
        }
        set
        {
            m_FirstName = value;
        }
    }

    [DataMember]
    public string LastName
    {
        get
        {
            return m_LastName;
        }
        set
        {
            m_LastName = value;
        }
    }
}
```

iar definitia importata va fi:

```
[DataContract]
public partial struct Contact
{
   string FirstNameField;
   string LastNameField;

   [DataMember]
   public string FirstName
   {
      get
      {
         return FirstNameField;
      }
      set
      {
         FirstNameField = value;
      }
   }

   [DataMember]
   public string LastName
   {
      get
      {
         return LastNameField;
      }
      set
      {
         LastNameField = value;
      }
   }
}
```

Observatie.

1. Atributul **DataMember** se aplica la o propietate *Read/Write*, in caz contrar se va genera exceptia **InvalidDataContractException**.
2. Nu trebuie aplicat atributul **DataMember** pe o proprietate si pe campurile private ce compun acea proprietate.
3. Atributul **DataMember** se aplica atat pe partea de client cat si pe partea de serviciu.

Contractul de date si atributul Serializable

Serviciul poate sa foloseasca un tip marcat numai cu atributul **Serializable**:

```
[Serializable]
struct Contact
{
   string m_FirstName;
   public string LastName;
}
```

Cand se importa metadata pentru acest tip, definitia importata va folosi atributul **DataContract**.

In acest caz fiecare camp care este serializabil va fi privit ca un camp adnotat cu **DataMember**.

Un tip adnotat numai cu **DataContract** nu poate fi serializat; trebuie sa folosim si atributul **Serializable** si **DataMember**.

Contracte de date compuse (agregate)

Cand se defineste un contract de date, trebuie aplicat atributul **DataMember** si pe campurile care au aplicat atributul **DataContract** in definitia lor.

Atentie la structura *Address* de mai jos.

```
[DataContract]
struct Address
{
   [DataMember]
   public string Street;

   [DataMember]
   public string City;

   [DataMember]
   public string State;

   [DataMember]
   public string Zip;
}

[DataContract]
struct Contact
{
   [DataMember]
   public string FirstName;

   [DataMember]
   public string LastName;

   [DataMember]
   public Address Address;
}
```

Cand se publica un contract de date compus, toate datele contractului vor fi publicate.

Exemplu :

```
[ServiceContract]
interface IContactManager
{
   [OperationContract]
   void AddContact(Contact contact);

   [OperationContract]
   Contact[] GetContacts( );
}
```

va include definitia structurii *Address*.

Evenimente pe contractul de date

Pentru serializare si deserializare sunt definite urmatoarele evenimente:

- **OnSerializing – inainte de ...**
- **OnSerialized – dupa ...**
- **OnDeserializing – inainte de ...**
- **OnDeserialized – dupa ...**

si atributele corespunzatoare.

```
[DataContract]
class MyDataContract
{
   [OnSerializing]
   void OnSerializing(StreamingContext context)
   {...}

   [OnSerialized]
   void OnSerialized(StreamingContext context)
   {...}

   [OnDeserializing]
   void OnDeserializing(StreamingContext context)
   {...}

   [OnDeserialized]
   void OnDeserialized(StreamingContext context)
   {...}
   //Data members
}
```

Fiecare metoda ce trateaza aceste evenimente trebuie sa aiba prototipul:

```
void <Method Name>(StreamingContext context);
```

Daca atributele sunt aplicate pe metode cu prototipuri eronate, WCF genereaza o exceptie.

StreamingContext

StreamingContext este o structura ce descrie sursa si destinatia unui stream de serializare dat si furnizeaza un context pentru apelant.

Constructor

Name	Description
StreamingContext(StreamingContextStates)	Initializes a new instance of the StreamingContext class with a given context state.
StreamingContext(StreamingContextStates, Object)	Initializes a new instance of the StreamingContext class with a given context state, and some additional information.

Proprietati

Name	Description
Context	Gets context specified as part of the additional context.
State	Gets the source or destination of the transmitted data.

Constructorul pentru **StreamingContext** foloseste o enumerare **StreamingContextStates** ce defineste o multime de "flags-uri" ce specifica contextul sursei sau destinatiei pentru stream, necesar in timpul serializarii. Enumerarea are atributul **FlagAttribute** ce permite o combinatie "bitwise" a valorilor membrilor sai.

Enumerarea StreamingContextStates

Member name	Description
CrossProcess	Specifies that the source or destination context is a different process on the same computer.
CrossMachine	Specifies that the source or destination context is a different computer.
File	Specifies that the source or destination context is a file. Users can assume that files will last longer than the process that created them and not serialize objects in such a way that deserialization will require accessing any data from the current process.
Persistence	Specifies that the source or destination context is a persisted store, which could include databases, files, or other backing stores. Users can assume that persisted data will last longer than the process that created the data and not serialize objects so that deserialization will require accessing any data from the current process.
Remoting	Specifies that the data is remoted to a context in an unknown location. Users cannot make any assumptions whether this is on the same computer.
Other	Specifies that the serialization context is unknown.
Clone	Specifies that the object graph is being cloned. Users can assume that the cloned graph will continue to exist within the same process and be safe to access handles or other references to unmanaged resources.
CrossAppDomain	Specifies that the source or destination context is a different AppDomain. For information on AppDomains, see Application Domains.
All	Specifies that the serialized data can be transmitted to or received from any of the other contexts.

Exemplu (MSDN) cu **StreamingContext**

```
using System;
  using System.Text;
```

```
using System.Runtime.Serialization.Formatters.Binary;
using System.Runtime.Serialization;
using System.IO;
using System.Security.Permissions;

namespace StreamingContextExample
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                SerializeAndDeserialize();
            }
            catch (System.Exception exc)
            {
                Console.WriteLine(exc.Message);
            }
            finally
            {
                Console.WriteLine("Press <Enter> to exit....");
                Console.ReadLine();
            }
        }
        static void SerializeAndDeserialize()
        {
            object myObject = DateTime.Now;

          // Create a StreamingContext that includes a
          // a DateTime.

            StreamingContext sc = new StreamingContext(
                StreamingContextStates.CrossProcess, myObject);
            BinaryFormatter bf = new BinaryFormatter(null, sc);
            MemoryStream ms = new MemoryStream(new byte[2048]);
            // memorare obiect (serializare)
            bf.Serialize(ms, new MyClass());

            // deserializare
           ms.Seek(0, SeekOrigin.Begin);
            MyClass f = (MyClass)bf.Deserialize(ms);
            Console.WriteLine("\t MinValue: {0} \n\t MaxValue: {1}",
                f.MinValue , f.MaxValue);
            Console.WriteLine("StreamingContext.State: {0}",
                sc.State);

            DateTime myDateTime = (DateTime)sc.Context;
            Console.WriteLine("StreamingContext.Context: {0}",
                myDateTime.ToLongTimeString());
        }
    }

    [Serializable]
    [SecurityPermission(SecurityAction.Demand,
                  SerializationFormatter = true)]
    class MyClass : ISerializable
    {
        private int minValue_value;
        private int maxValue_value;
```

```
            public MyClass()
            {
                minValue_value = int.MinValue;
                maxValue_value = int.MaxValue;
            }

            public int MinValue
            {
                get { return minValue_value; }
                set { minValue_value = value;}
            }

            public int MaxValue
            {
                get { return maxValue_value; }
                set { maxValue_value = value; }
            }

            void ISerializable.GetObjectData(SerializationInfo si,
                        StreamingContext context)
            {
                si.AddValue("minValue", minValue_value);
                si.AddValue("maxValue", maxValue_value);
            }

            protected MyClass(SerializationInfo si,
                        StreamingContext context)
            {
              minValue_value = (int)si.GetValue("minValue", typeof(int));
              maxValue_value = (int)si.GetValue("maxValue", typeof(int));
            }
        }
    }
```

Exemplu de tratare eveniment pe deserializare terminata.

```
[DataContract]
class MyDataContract
{
   IDbConnection m_Connection;

   [OnDeserialized]
   void OnDeserialized(StreamingContext context)
   {
      m_Connection = new SqlConnection(...);
   }
   /* Data members */
}
```

Ierarhia contractului de date

In cazul unei ierarhii de clase folosite drept contracte de date, fiecare clasa trebuie adnotata cu atributul **DataContract**, in caz contrar se genereaza o exceptie.

```
[DataContract]
class Contact
{
   [DataMember]
   public string FirstName;

   [DataMember]
   public string LastName;
}

[DataContract]
class Customer : Contact
{
   [DataMember]
   public int OrderNumber;
}
```

Cand se exporta o ierarhie de contracte de date, metadata contine ierarhia.

Tipuri cunoscute. Atributul `KnownTypeAttribute`

In C# putem substitui o subclasa pentru o clasa de baza, in WCF nu.

KnownTypeAttribute permite construirea de subclase pentru contractul de date.

Exemplu

```
[DataContract]
class Customer : Contact
{
   [DataMember]
   public int OrderNumber;
}

[ServiceContract]
interface IContactManager
{
   // Aici nu se accepta un obiect Customer
   [OperationContract]
   void AddContact(Contact contact);

   // Aici nu pot fi returnate obiecte Customer:
   [OperationContract]
   Contact[] GetContacts( );
}
```

Motivul este legat de serializare si deserializare. Nu se stie modul cum sa se contruiasca o instanta pentru *Customer*.

Solutia consta in a spune WCF-ului in mod explicit ce inseamna clasa *Customer* folosind atributul **KnownTypeAttribute** definit astfel:

```
[AttributeUsage(AttributeTargets.Struct|AttributeTargets.Class,
                AllowMultiple = true)]
public sealed class KnownTypeAttribute : Attribute
{
   public KnownTypeAttribute(Type type);
```

```
    //More members
}
```

Acest atribut ne permite sa proiectam subclase pentru contractul de date. Atributul se aplica la clasa de baza.

```
[DataContract]
[KnownType(typeof(Customer))]
class Contact
{...}

[DataContract]
class Customer : Contact
{...}
```

"Tipul Contact este un tip cunoscut de tipul Customer".

Pe partea de server, atributul **KnownType** afecteaza ***toate*** contractele si operatiile ce folosesc clasa de baza si permite subclase in locul clasei de baza.

In metadata apare si subclasa.

Atributul `ServiceKnownTypeAttribute`

Acest atribut se aplica operatiilor (si nu numai) si indica ca numai acele operatii pot accepta subclase.

Este definit astfel:

```
[AttributeUsage(AttributeTargets.Interface|
                AttributeTargets.Method   |
                AttributeTargets.Class,
                AllowMultiple = true)]
public sealed class ServiceKnownTypeAttribute : Attribute
{
   public ServiceKnownTypeAttribute(Type type);
   //More members
}
```

Observatie
Atentie la tipurile ce pot fi adnotate cu acest atribut.

Exemplu:

```
[DataContract]
class Contact
{...}

[DataContract]
class Customer : Contact
{...}

[ServiceContract]
interface IContactManager
{
   [OperationContract]
```

```
    [ServiceKnownType(typeof(Customer))]
    void AddContact(Contact contact);

    [OperationContract]
    Contact[] GetContacts( );
}
```

Cand atributul se aplica la nivel de contract, toate operatiile din acel contract pot accepta subclase.

```
[ServiceContract]
[ServiceKnownType(typeof(Customer))]
interface IContactManager
{
    [OperationContract]
    void AddContact(Contact contact);

    [OperationContract]
    Contact[] GetContacts( );
}
```

Observatie
Nu trebuie aplicat acest atribut la nivel de serviciu pentru ca nu are efect asupra contractelor de date definite separat.

ServiceKnownType are proprietatea Multiple=true, deci poate fi aplicat de mai multe ori pe aceeasi clasa.

```
[DataContract]
class Contact
{...}

[DataContract]
class Customer : Contact
{...}

[DataContract]
class Person : Contact
{...}

[ServiceContract]
[ServiceKnownType(typeof(Customer))]
[ServiceKnownType(typeof(Person))]
interface IContactManager
{...}
```

Nu functioneaza ceea ce stim de la mostenire. Trebuie sa adaugam tot lantul de derivare.

```
[DataContract]
class Contact
{...}

[DataContract]
class Customer : Contact
{...}

[DataContract]
class Person : Customer
{...}
```

```
[ServiceContract]
[ServiceKnownType(typeof(Customer))]
[ServiceKnownType(typeof(Person))]
interface IContactManager
{...}
```

Configurare

```
<system.runtime.serialization>
   <dataContractSerializer>
      <declaredTypes>
         <add type = "Contact,Host,Version=1.0.0.0,Culture=neutral,
                      PublicKeyToken=null">
            <knownType type = "Customer,MyClassLibrary,Version=1.0.0.0,
                               Culture=neutral,PublicKeyToken=null"/>
         </add>
      </declaredTypes>
   </dataContractSerializer>
</system.runtime.serialization>
```

Observatie
Daca un tip este intern in alt assembly, fisierele de configurare constituie singura modalitate de a le face cunoscute in server si client.

Object si Interfete

Tipul de baza al unui contract poate fi o interfata:

```
interface IContact
{
   string FirstName
   {get;set;}
   string LastName
   {get;set;}
}

[DataContract]
class Contact : IContact
{...}
```

Se poate folosi aceasta interfata in contractul de serviciu sau ca data membru in contracte de date atata timp cat folosim atributul **ServiceKnownType** in proiectarea tipului de data:

```
[ServiceContract]
[ServiceKnownType(typeof(Contact))]
interface IContactManager
{
   [OperationContract]
   void AddContact(IContact contact);

   [OperationContract]
   IContact[] GetContacts( );
}
```

Atributul **KnownType** nu poate fi aplicat pe interfete pentru ca interfetele nu sunt incluse in metadata exportata.

Contractul exportat va fi bazat pe **object** si va include subclasa contractului de date sau structura fara derivare:

```
//Imported definitions:
[DataContract]
class Contact
{...}

[ServiceContract]
public interface IContactManager
{
    [OperationContract]
    [ServiceKnownType(typeof(Contact))]
    [ServiceKnownType(typeof(object[]))]
    void AddContact(object contact);

    [OperationContract]
    [ServiceKnownType(typeof(Contact))]
    [ServiceKnownType(typeof(object[]))]
    object[] GetContacts( );
}
```

Definitia importata va avea **ServiceKnownType** aplicat la nivel de operatie, chiar daca la creare a fost aplicat la nivel de contract.

Fiecare operatie va include o reuniune a tuturor atributelor **ServiceKnownType** cerute de toate operatiile.

```
[DataContract]
class Contact
{...}

[ServiceContract]
public interface IContactManager
{
   [OperationContract]
   [ServiceKnownType(typeof(Contact))]
   void AddContact(object contact);

   [OperationContract]
   [ServiceKnownType(typeof(Contact))]
   object[] GetContacts( );
}
```

Daca avem definitia interfetei pe partea de client atunci o putem folosi in locul lui object.

```
[DataContract]
class Contact : IContact
{...}

[ServiceContract]
public interface IContactManager
{
    [OperationContract]
```

```
    [ServiceKnownType(typeof(Contact))]
    void AddContact(IContact contact);

    [OperationContract]
    [ServiceKnownType(typeof(Contact))]
    IContact[] GetContacts( );
}
```

DataMember(IsRequired=true/false)]

In informatia serializata / deserializata membrii respectivi trebuie sa existe.

[DataMember(IsRequired=true)]

Un membru al tipului adnotat cu **DataMember** poate sa nu existe in procesul de serializare / deserializare – se considera valoarea implicita atribuita tipurilor preconstruite sau *null* pentru tipuri referinta.

```
[DataContract]
struct Contact
{
   [DataMember]
   public string FirstName;

   [DataMember]
   public string LastName;

   [DataMember(IsRequired = true)]
   public string Address;
}
```

Cand sunt exportate tipuri custom serializabile, toti membrii tipului sunt adnotati cu **[DataMember(IsRequired=true)].**

```
[Serializable]
struct Contact
{
   public string FirstName;
   public string LastName;
}
```

si metadata este:

```
[DataContract]
struct Contact
{
   [DataMember(IsRequired = true)]
   public string FirstName
   {get;set;}

   [DataMember(IsRequired = true)]
   public string LastName
   {get;set;}
}
```

Atributul [**OptionalField**] are ca efect export metadata fara **IsRequired=true**.

```
[Serializable]
struct Contact
{
   public string FirstName;

   [OptionalField]
   public string LastName;
}
```

iar metadata este:

```
[DataContract]
struct Contact
{
   [DataMember(IsRequired = true)]
   public string FirstName
   {get;set;}

   [DataMember]
   public string LastName
   {get;set;}
}
```

Enumerari

Enumerarile sunt totdeauna serializabile. Nu trebuie aplicat atributul [**DataContract**]. Toate valorile din enum vor fi incluse in mod implicit in contract.

```
enum ContactType
{
   Customer,
   Vendor,
   Partner
}

[DataContract]
struct Contact
{
   [DataMember]
   public ContactType ContactType;

   [DataMember]
   public string FirstName;

   [DataMember]
   public string LastName;
}
```

Daca se doreste ca numai anumite valori sa fie prinse in contract se foloseste atributul [**EnumMember**] pe valoarea respectiva.

Definitia atributului este:

```
[AttributeUsage(AttributeTargets.Field,Inherited = false)]
public sealed class EnumMemberAttribute : Attribute
{
```

```
    public string Value
    {get;set;}
}

[DataContract]
enum ContactType
{
    [EnumMember]
    Customer,

    [EnumMember]
    Vendor,

    // Nu va fi parte a contractului
    Partner
}
```

Alias pentru anumite valori:

```
[DataContract]
enum ContactType
{
    [EnumMember(Value = "MyCustomer")]
    Customer,

    [EnumMember]
    Vendor,

    [EnumMember]
    Partner
}
```

si va reprezenta in fapt:

```
enum ContactType
{
    MyCustomer,
    Vendor,
    Partner
}
```

DataSet si DataTable

DataSet si **DataTable** sunt definite ca tipuri *serializabile* in .NET.

DataRow nu e *serializabil*.

```
[Serializable]
public class DataSet : ...
{...}

[Serializable]
public class DataTable : ...
{...}
```

In concluzie putem scrie ceva de genul:

```
[DataContract]
struct Contact
{...}

[ServiceContract]
interface IContactManager
{
   [OperationContract]
   void AddContact(Contact contact);

   [OperationContract]
   void AddContacts(DataTable contacts);

   [OperationContract]
   DataTable GetContacts( );
}
```

Cand se importa definitia pentru contractul de mai sus, proxy va contine definitia pentru DataTable.

Array in loc de tabele

Array contine *object*, deci putem pune orice avand grija la conversii. Se poate scrie o metoda ce transforma multimea inregistrarilor unei tabele intr-un array (Incercati!).

Colectii

Colectia = tip ce suporta interfetele **IEnumerable** sau **IEnumerable<T>.**

Un contract de date poate include o colectie ca date membru, sau un contract de serviciu poate defini operatii ce intercationeza cu colectia in mod direct.

WCF ofera reguli diferite pentru lucrul cu colectii.

Cand definim o operatie pe un serviciu ce folosesc urmatoarele interfete:

- **IEnumerable<T> ;**
- **IList<T> ;**
- **ICollection<T>.**

Reprezentarea in retea va folosi totdeauna un *array*.

Exemplu :

```
[ServiceContract]
interface IContactManager
{
   [OperationContract]
   IEnumerable<Contact> GetContacts( );
   ...
}
class ContactManager : IContactManager
{
```

```
   List<Contact> m_Contacts = new List<Contact>( );

   public IEnumerable<Contact> GetContacts( )
   {
      return m_Contacts;
   }
   ...
}
```

va fi exportat ca:

```
[ServiceContract]
interface IContactManager
{
   [OperationContract]
   Contact[] GetContacts( );
}
```

Colectii concrete - nu o interfata

Daca colectia din contract nu este o interfata si este o colectie serializabila (adnotata cu atributul **Serializable** si nu cu **DataContract**), WCF poate normaliza colectia la un array de tipul colectiei, furnizand metoda **Add**() cu una din semnaturile:

```
public void Add(object obj); //Collection uses IEnumerable
public void Add(T item);     //Collection uses IEnumerable<T>
```

Exemplu

```
[ServiceContract]
interface IContactManager
{
   [OperationContract]
   void AddContact(Contact contact);

   [OperationContract]
   List<Contact> GetContacts( );
}
```

Clasa **List** este definita astfel:

```
public interface ICollection<T> : IEnumerable<T>
{...}
public interface IList<T> : ICollection<T>
{...}
[Serializable]
public class List<T> : IList<T>
{
   public void Add(T item);
   //More members
}
```

Pentru ca e o colectie valida si are metoda **Add**(), reprezentarea contractului va fi:

```
[ServiceContract]
interface IContactManager
```

```
{
   [OperationContract]
   void AddContact(Contact contact);

   [OperationContract]
   Contact[] GetContacts( );
}
```

List<Contacts> este transferata drept *Contact[].*
Serviciul poate sa returneze List<Contacts>, si clientul sa lucreze cu un array.

```
//////////////////////////// Service Side ///////////////////////////////

[ServiceContract]
interface IContactManager
{
   [OperationContract]
   void AddContact(Contact contact);

   [OperationContract]
   List<Contact> GetContacts( );
}

//Service implementation
class ContactManager : IContactManager
{
   List<Contact> m_Contacts = new List<Contact>( );

   public void AddContact(Contact contact)
   {
      m_Contacts.Add(contact);
   }

   public List<Contact> GetContacts( )
   {
      return m_Contacts;
   }
}

//////////////////////////// Client Side ///////////////////////////////
// proxy ...
[ServiceContract]
interface IContactManager
{
   [OperationContract]
   void AddContact(Contact contact);

   [OperationContract]
   Contact[] GetContacts( );
}
public partial class ContactManagerClient :
            ClientBase<IContactManager>, IContactManager
{
   public Contact[] GetContacts( )
   {
      return Channel.GetContacts( );
   }
}

//Client code
```

```
ContactManagerClient proxy = new ContactManagerClient( );
Contact[] contacts = proxy.GetContacts( );
proxy.Close( );
```

Colectii personalizate

Orice colectie personalizata poate fi transferata ca un array.

Colectia trebuie sa fie serializabila.

```
//////////////////////////// Service Side ////////////////////////////////
[Serializable]
public class MyCollection<T> : IEnumerable<T>
{
   public void Add(T item)
   {}

   IEnumerator<T>  IEnumerable<T>.GetEnumerator( )
   {...}
   //Rest of the implementation
}
[ServiceContract]
interface IMyContract
{
   [OperationContract]
   MyCollection<string> GetCollection( );
}

//////////////////////////// Client Side ////////////////////////////////
[ServiceContract]
interface IMyContract
{
   [OperationContract]
   string[] GetCollection( );
}
```

Contract de date pentru colectii

Atributul `CollectionDataContractAttribute` este definit astfel:

```
[AttributeUsage(AttributeTargets.Struct|AttributeTargets.Class,
Inherited = false)]
public sealed class CollectionDataContractAttribute : Attribute
{
   public string Name
   {get;set;}
   public string Namespace
   {get;set;}
   //More members
}
```

Acest atribut nu face colectia serializabila, doar o expune clientului ca o lista generica inlantuita.

Exemplu

```
[CollectionDataContract(Name = "MyCollectionOf{0}")]
public class MyCollection<T> : IEnumerable<T>
{
   public void Add(T item)
   {}

   IEnumerator<T>  IEnumerable<T>.GetEnumerator( )
   {...}
   //Rest of the implementation
}
```

si definitia in serviciu

```
[ServiceContract]
interface IContactManager
{
   [OperationContract]
   void AddContact(Contact contact);

   [OperationContract]
   MyCollection<Contact> GetContacts( );
}
```

Pe partea de client importul arata astfel:

```
[CollectionDataContract]
public class MyCollectionOfContact : List<Contact>
{}

[ServiceContract]
interface IContactManager
{
   [OperationContract]
   void AddContact(Contact contact);

   [OperationContract]
   MyCollectionOfContact GetContacts( );
}
```

Observatie

La incarcarea serviciului se verifica prezenta metodei **Add**() si **IEnumerable** sau **IEnumerable<T>.** Lipsa acestora va genera o exceptie **InvalidDataContractException**.

Nu se poate aplica **DataContract** si **CollectionDataContract** impreuna.

Referentierea colectiei

SvcUtil.exe are switch **/collectionType** (sau **/ct**) pentru a ne permite sa facem referire la o colectie particulara dintr-un assembly, pentru a extrage metadata pe partea de client. Trebuie specificata locatia assembly-ului si bineinteles assembly trebuie sa fie disponibil clientului.

De exemplu, serviciul ar putea defini urmatorul contract ce foloseste colectia **Stack<T>:**

```
[ServiceContract]
interface IContactManager
{
   [OperationContract]
   void AddContact(Contact contact);

   [OperationContract]
   Stack<Contact> GetContacts( );
}
```

Linia de comanda pentru SvcUtil.exe este:

```
SvcUtil http://localhost:8000/
        /r:C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.dll
        /ct:System.Collections.Generic.Stack'1
```

Rezultatul definitiei contractului pentru client va fi (se pastreaza definitia lui **Stack<T>):**

```
[ServiceContract]
interface IContactManager
{
   [OperationContract]
   void AddContact(Contact contact);

   [OperationContract]
   Stack<Contact> GetContacts( );
}
```

Dictionare

Dictionarele au propria reprezentare in WCF.

1. Daca dictionarul este o colectie serializabila ce suporta interfata **IDictionary**, atunci va fi expus ca **Dictionary<object, object>.**

```
[Serializable]
public class MyDictionary : IDictionary
{...}

[ServiceContract]
interface IContactManager
{
   ...
   [OperationContract]
   MyDictionary GetContacts( );
}
```

va fi expus astfel :

```
[ServiceContract]
interface IContactManager
{
    ...
    [OperationContract]
    Dictionary<object,object> GetContacts( );
}
```

2. Daca colectia serializabila suporta interfata **IDictionary<K,T>** reprezentarea va fi ca **Dictionary<K,T>:**

```
[Serializable]
public class MyDictionary<K,T> : IDictionary<K,T>
{...}

[ServiceContract]
interface IContactManager
{
    ...
    [OperationContract]
    MyDictionary<int,Contact> GetContacts( );
}
```

va fi expus astfel:

```
[ServiceContract]
interface IContactManager
{
    ...
    [OperationContract]
    Dictionary<int,Contact> GetContacts( ); }
```

3. Daca folosim o colectie derivata din **IDictionary**, aceasta va fi transferata ca o subclasa a respectivei reprezentari.

```
[CollectionDataContract]
public class MyDictionary : IDictionary
{...}

[ServiceContract]
interface IContactManager
{
    ...
    [OperationContract]
    MyDictionary GetContacts( );
}
```

va avea reprezentarea

```
[CollectionDataContract]
public class MyDictionary : Dictionary<object,object>
{}

[ServiceContract]
interface IContactManager
```

```
{
   ...
   [OperationContract]
   MyDictionary GetContacts( );
}
```

in timp ce colectia generica:

```
[CollectionDataContract]
public class MyDictionary<K,T> : IDictionary<K,T>
{...}

[ServiceContract]
interface IContactManager
{
   ...
   [OperationContract]
   MyDictionary<int,Contact> GetContacts( );
}
```

va fi publicata in metadata ca:

```
[CollectionDataContract]
public class MyDictionary : Dictionary<int,Contact>
{}

[ServiceContract]
interface IContactManager
{
   ...
   [OperationContract]
   MyDictionary GetContacts( );}
```