Kurt Claeys is werkzaam als .NET Architect bij Ordina Belgium

WCF Reliable en Transacted Messaging

ENTERPRISE APPLICATIONS BOUWEN GEBASEERD OP WCF-CONTRACTEN

Windows Communication Framework (WCF) is een omgeving waarin je services kunt bouwen die door middel van een uniform programmeermodel aanspreekbaar zijn vanuit verschillende communicatietechnologieën. Deze services gebruiken SOAP-messages om te communiceren. Enterprise applications stellen een aantal hoge eisen aan deze gegevensoverdracht en verwachten dat deze betrouwbaar, transactioneel en asynchroon verlopen. In dit artikel gaat de auteur dieper in op enkele technieken die hiervoor nodig zijn.

Windows Communication Foundation verzamelt de bestaande technieken die nodig zijn voor het bouwen van distributed applications zoals webservices (.asmx), .NET remoting, MSMQ, WS-Enhancements 3.0 en Enterprise Services. Met behulp van Windows Communication Foundation is het onder meer mogelijk:

- services reliable te maken, waardoor de messages van client naar server gegarandeerd en in de correcte volgorde aankomen.
- services statefull te maken op basis van sessies die door clients zijn opgestart.
- services transactioneel op te bouwen, waardoor verschillende methods kunnen participeren in één transactie.
- het opzetten van services onafhankelijk te maken van technische implementatie-elementen als hosting en transportprotocollen.
- servicemethods asynchroon aan te roepen.

WCF-services zijn opgebouwd rond het concept van Address, Binding en Contract; kortweg ABC genoemd. Samen vormen ze een endpoint. Zowel de client als de service hebben dit endpoint. Hierbij maakt enkel het Contract deel uit van de applicatie, de Binding en het Address bevinden zich buiten de context van de applicatie en zijn eenvoudig configureerbaar. Door dit model zijn services onafhankelijk van hun technische implementatie wat adressering en transportmechanismes betreft. Het zijn vooral de contracten en hun configuratie die de basis vormen voor de nodige vereisten van enterprise applications. Het contract bepaalt wat de service doet. Hierin is beschreven wat de namen zijn van de methods en de types van hun parameters. Daarnaast kunnen in het contract zaken geconfigureerd zijn die voor de client van belang zijn om te weten hoe de service zich gedraagt, zoals de implementatie van transacties en ordered messaging. Er zijn verschillende types van contracten, elk met hun specifieke betekenis; zie codevoorbeeld 1.

ServiceContract: is een contract dat uit een .NET-interface bestaat, waarin de mogelijke operaties van de service beschreven zijn. De interface bevat geen implementatie van de werking van deze operaties. Naast deze interface kan door middel van het ServiceContract ook het gedrag van de service bepaald worden. Het contract vormt de basis voor de automatische generatie van het WSDL-document dat door Visual Studio wordt gebruikt voor het genereren van de nodige proxies. Een ServiceContract kan verscheidene OperationContracts bevatten. Voor elke functie die via de service aanspreekbaar moet zijn, is er een OperationContract. Ook hierin kunnen extra attributen opgegeven worden om de werking van de specifieke operatie verder te bepalen.

MessageContract: hiermee kan gedetailleerder bepaald worden hoe de eigenlijke SOAP-message er zal uitzien. Het laat toe om aan te duiden welke velden tot de header en welke velden tot de body behoren. Het is een geavanceerde techniek die enkel nodig is om een extra controle te hebben over de SOAP-message.

DataContract: is een contract dat vastlegt wat de structuur is van een boodschap die als parameter of returnwaarde aan een operatie is verbonden. Het kan bijvoorbeeld aanwijzingen bevatten over de naamgeving van de velden in de structuur. Het beschrijft concreet hoe de XML-serialization van de data zal gebeuren. Vanuit deze structuur zal de XSD gegenereerd worden die bruikbaar is voor de buitenwereld.

Publiceren van een service

Door middel van WCF is men niet langer meer gebonden aan IIS als host voor het publiceren van services. Door de onafhankelijkheid van Binding en Address zijn er verschillende mogelijkheden de service te hosten. Het is de bedoeling dat de klasse die een ServiceContract heeft aan de WCF-infrastrucuur bekend wordt gemaakt.

In IIS: uiteraard blijft IIS kandidaat om de service te hosten; dit door een file met extensie .SVC in een webapplicatie-directory te plaatsen. Deze file bevat enkel een verwijzing naar de klasse met het contract. Door de plaats van de file in de directorystructuur is hiermee ook de adressering van de service bepaald; zie codevoorbeeld 2.

In een eigen applicatie: het is ook mogelijk de service vanuit een eigen applicatie te publiceren. Dit kan bijvoorbeeld vanuit een consoleapplicatie, maar in werkelijkheid zal hiervoor meestal een windows-service gebruikt worden. De service is slechts bereikbaar als de host waarin ze draait actief is. Dit is met een consoleapplicatie niet steeds zo, terwijl een windows-service altijd zijn werk doet. Hiervoor maakt men gebruik van de ServiceHost-klasse die het type van de serviceklasse nodig heeft in zijn constructor. De URI waarop de service zal gehost worden is configureerbaar in de app.config op een speciaal voor WCF voorziene plaats; zie codevoorbeeld 3a en 3b.

```
[ServiceContract()]
public interface IKlantenService
{
   [OperationContract]
   void Method1(ClientData TheParam);

   [OperationContract]
   ClientData Method2(string TheParam);

   [OperationContract(IsOneWay = true)]
   void Method3(string TheParam);
}

[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerSession)]
public class KlantenService : IKlantenService
{
   public void Method1(ClientData TheParam)
   {
      //implementatie
   }
   public ClientData Method2(string TheParam)
   {
      //implementatie
   }
   public void Method3(string TheParam)
   {
      //implementatie
   }
}

[DataContract(Namespace = "http://mijnklantenschema.mijnfirma.be",
      Name = "MijnKlanten")]
public class ClientData
{
   string _KlantID;

   [DataMember(Name = "CustID")]
   public string CustIdentification
   {
      get { return _KlantID; }
      set { _KlantID = value; }
   }

   private string _BTWNummer;

   [DataMember(Name = "VAT")]
   public string VATNumber
   {
      get { return _BTWNummer; }
      set { _BTWNummer = value; }
   }
}
```
Codevoorbeeld 1.WCF-contracten

ServiceBehavior en InstanceContextMode

De servicebehavior schrijft voor hoe een service zich op de server gedraagt tussen het aanroepen van de verschillende methods vanuit de client. Dit gedrag is van belang om de methods in de service een bepaalde werkwijze op te leggen hoe ze met state moeten omgaan. De service is eigenlijk een klasse die door het WCF-framework geïnstantieerd wordt op de server. Hoe en wanneer deze klasse wordt geïnstantieerd is afhankelijk van de parameter InstanceContextMode van het attribuut ServiceBehavior op de implementatie van de klasse. Deze kan zijn: Single, PerCall en PerSession; zie afbeelding 1.

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.)]
  PerCall
  PerSession
  Single
```
Afbeelding 1. De mogelijke InstanceContextModes

Single: hierdoor zal iedere client (ook als die vanuit verschillende machines komen) aan één en dezelfde instantiatie van de klasse op de server worden gekoppeld. Deze instantiatie wordt eenmalig uitgevoerd bij het opstarten van de hostapplicatie en blijft dus bestaan, ook als geen enkele client hier nog naar verwijst. Hierdoor blijft de service zijn state behouden en is deze state dan ook volledig bereikbaar vanuit andere clients; zie afbeelding 2.

PerCall: voor elke aanroep die een client doet, zal de serviceklasse door WCF op de server geïnstantieerd en toegekend worden aan de client. Na het aflopen van de method in de service zal deze instantiatie verwijderd worden. Hierdoor is deze service stateless en totaal geïsoleerd van andere clients; zie afbeelding 3.

PerSession: er vindt een instantiatie plaats van de klasse in een sessie die door de client is opgestart en deze blijft in het geheugen zolang de sessie duurt. De client bepaalt dus de levensduur van de instantiatie op de server. Hierdoor is de state alleen beschikbaar voor de client die specifiek de sessie heeft opgestart en bereikbaar vanuit verschillende methods die deel uitmaken van de sessie. Er is een sessie voor elk van de gebruikte referentievariabelen op de client, waardoor ze allemaal aanspraak kunnen maken op een eigen state; zie afbeelding 4.

De InstanceContextMode bepaalt eigenlijk wanneer de klasse wordt geïnstantieerd. Bij Single is dit bij het opstarten van de host. Bij PerCall is dit bij elke aanroep. Bij PerSession is dit bij het aanroepen van de eerste method en dit per instantie van de classe.

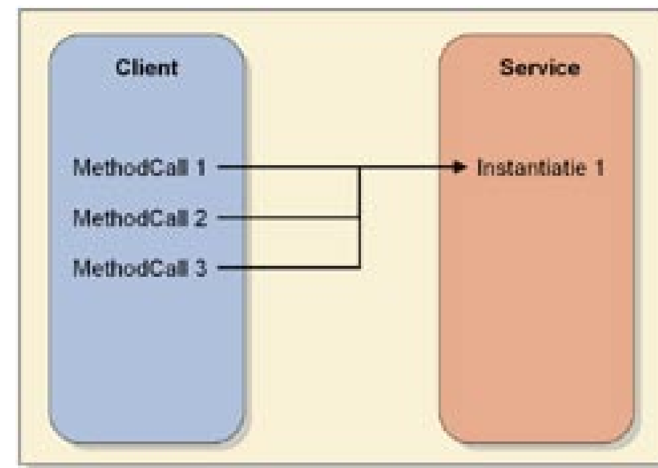
Ordered Reliable Sessions

Door met sessions te werken kan de state worden bijgehouden tussen verschillende methods in de sessie binnen de scope van de instantie van de service class, waarbij deze state toch geïsoleerd blijft voor andere clients. Daarnaast is de logica van een service dikwijls zodanig opgebouwd dat een bepaalde method uitgevoerd dient te worden voordat men de andere methods mag aanroepen. WCF gebruikt hiervoor Ordered Reliable Sessions. Het maken van zo'n sessiegeoriënteerde service is eenvoudig te verwezenlijken met WCF. Op het ServiceContract kan bepaald worden dat de client een sessie moet gebruiken door de parameter SessionMode op Required te zetten. Door middel van het OperationContract kan bepaald worden of de method een sessie start, een sessie stopt of gewoon deel uitmaakt van een sessie. Dit wordt aangegeven door de Booleaanse parameters IsInitiating en IsTerminating. De levensduur van een sessie wordt dus door de client bepaald. De sessie start bij het aanroepen van de Initiating-method en de sessie eindigt bij het aanroepen van de Terminating-method. Ook kan het zijn dat door een onstabiel netwerk de opeenvolgende calls naar methods niet in dezelfde volgorde op de server aankomen zoals de client ze heeft opgestart. Hierdoor zou de service verkeerde beslissingen kunnen nemen. Om dit te voorkomen maakt Ordered Reliable Messaging gebruik van het attribuut DeliveryRequirements om aan te geven dat bij implementatie door middel van configuratie een WS-RM compatible binding moet gebruikt worden. Bij een correcte configuratie is het resultaat dat de calls in dezelfde volgorde verwerkt zullen worden als ze zijn verzonden. Zoniet zal reeds bij het laden van de configuratie een exception voorkomen.; zie codevoorbeeld 4.

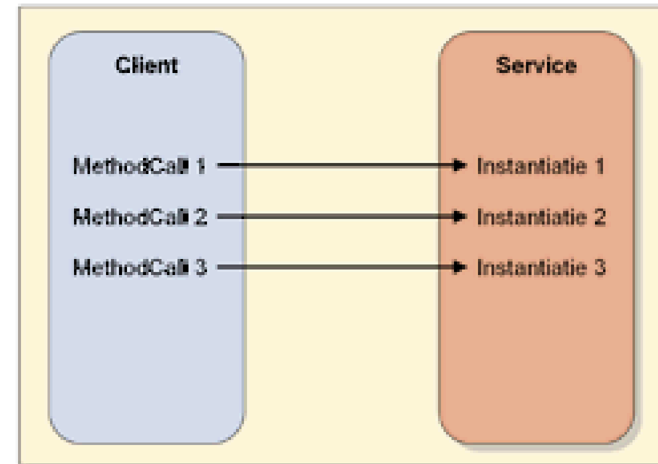
```
<%@ServiceHost language="c"# Debug="true"Service="MijnNameSpace.MijnService">
```
Codevoorbeeld 2. CF Service-file

```
ServiceHost MijnServiceHost = new ServiceHost(typeof(MijnService));

MijnServiceHost.Open();

Console.WriteLine("De service is actief ...");
Console.ReadKey();

MijnServiceHost.Close();
```
Codevoorbeeld 3a. Het hosten van een service

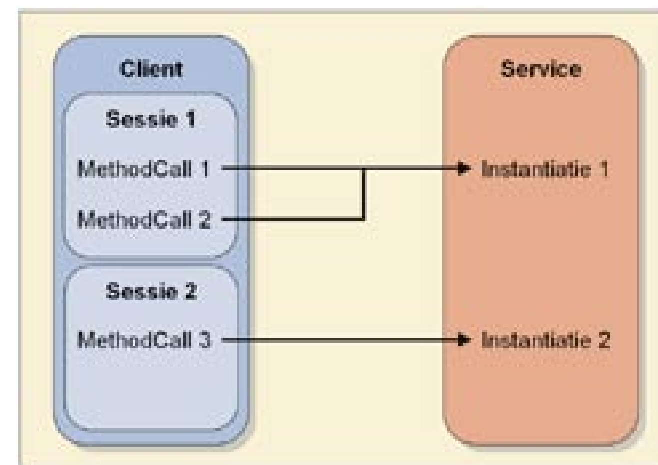
```
<service name="MijnService" behaviorConfiguration="MyServiceTypeBehaviors">
  <endpoint address="" contract="IMijnService" binding="basicHttpBinding"/>
  <host>
    <baseAddresses>
      <add baseAddress="http://localhost:8000/MijnService/Service"/>
    </baseAddresses>
  </host>
</service>
```
Codevoorbeeld 3b. Configuratie van de URI



Afbeelding 2. ServiceBehavior Single



Afbeelding 3. ServiceBehavior PerCall



Afbeelding 4. ServiceBehavior PerSession

Transacties in WCF

Een transactie is een programmeerconstructie die aangeeft dat verscheidene opeenvolgende wijzigingen in een databasetabel als een geheel moet worden gezien. Mocht er zich tussen de verschillende aanroepen een fout voordoen (wat in een exception resulteert), dan worden de wijzigingen die voor de fout al zijn uitgevoerd teruggedraaid (rollback). Transacties zorgen er dus voor dat de database in een consistente state blijft, ook na het voorkomen van fouten. Meestal bevatten transacties mutaties in database-tables, maar het is ook mogelijk variabelen transactioneel te gebruiken. Als voorbeeld nemen we een debet- en creditoperatie op bankrekeningen. Stel dat er zich na het uitvoeren van de debetactie een fout voordoet waardoor de creditactie onuitgevoerd blijft, dan is het uiteraard de bedoeling dat ook deze debetactie als onuitgevoerd zal worden beschouwd. De transactie kan in WCF worden opgestart op initiatief van de client die de service aanroept. Door eenvoudigweg de calls naar de service in een transactionscope te plaatsen, komt de service te weten dat de client een transactie heeft opgestart. De service inlists in de flow van de transactie, waardoor de mutaties op de databasetabel slechts effectief worden als de client te kennen geeft dat er niets is fout gegaan. Hiervoor dient de client gebruik te maken van de TransactionScope Class (te vinden in de System.Transactions-namespace). Dit is een klasse die nieuw is voor .NET2.0 en waarmee het mogelijk is een transactie die opgestart is als een lokale transactie automatisch te laten promoveren naar een distributed transactie. Door de Complete-method uit te voeren, geeft de client aan dat de transactie afgerond is; zie codevoorbeeld 5.

Het gebruik van transacties die door de clients zijn opgestart, heeft als voordeel dat de verantwoordelijkheid om een transactie als definitief te beschouwen bij de client kan liggen, maar moet met de

```
DeliveryRequirements(RequireOrderedDelivery = true)]
[ServiceContract(SessionMode = SessionMode.Required)]
public interface IKlantenService
{
   [OperationContract(IsInitiating = true)]
   void Method1(string TheParam);

   [OperationContract(IsTerminating = true)]
   string Method2(string TheParam);
}

[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerSession)]
public class KlantenService : IKlantenService
{
   //implementatie
}
```
Codevoorbeeld 4. Attributen voor Ordered Reliable Sessions

```
WCFRef.KlantenServiceClient S = new WCFRef.KlantenServiceClient();

using (TransactionScope scope = new TransactionScope(TransactionScopeOption.RequiresNew))
{
   try
   {
      S.Debet("1", 100);
      //implementatie die mogelijks fout loopt
      S.Credit("2", 100);
      scope.Complete();
   }
   catch (System.Exception ex)
   {
   //Vang de fout op
   }
}
```
Codevoorbeeld 5. Transacties vanuit de client

nodige voorzichtigheid aangewend worden daar het een ernstige impact heeft op performance en schaalbaarheid. Door het feit dat de client zelf kan beslissen wat hij als een transactie beschouwt, kunnen we de services heel generiek maken. De service zal zijn werk doen in situaties waar geen client-transactie is opgestart, maar ook wanneer de client wel een transactie heeft opgestart. Ook als de client geen transactie meegeeft, kan de method zodanig gedeclareerd worden dat er toch een transactie wordt gecreëerd. Uiteraard zal deze transactie zich enkel en alleen tijdens de uitvoering van die method (en eventuele methods die van hieruit worden opgeroepen) afspelen. Ook kan de method gedeclareerd worden, zodat een transactie vanuit de client is vereist. Als deze niet aanwezig is, zal de service onmiddellijk een exception gooien. De servicemethods kunnen geconfigureerd worden met elementen die aangeven hoe te reageren op het al dan niet aanwezig zijn van een transactie op de client. Dit kan door middel van het attribuut OperationBehavior op de implementatie van de method. Hier zijn twee parameters van belang; zie tabel 1.
Daarnaast bestaat het attribuut TransactionFlow om aan te geven of de client verplicht is een transactie wel of niet te gebruiken. Dit attribuut moet op de method in de interface worden geplaatst. De parametermogelijkheden voor dit attribuut zijn: Allowed (de client mag, maar hoeft geen transactie mee te geven), Mandatory (de client is verplicht om een transactie mee te geven), NotAllowed (de client mag geen transactie meegeven); zie codevoorbeeld 6.

Onafhankelijkheid van hosting en transportprotocol

In de bestaande distributed services-technologieën zoals .asmx-webservices, .NET remoting en MSMQ diende de ontwikkelaar in het model van één van deze technologieën te programmeren. Hierdoor werd gebruik gemaakt van een voor elk model specifieke API en configuratietechniek en bestond er een incompatibiliteit tussen deze technieken. Het was bijvoorbeeld niet logisch om een .NET remoting client met een .asmx-webservice te laten spreken. Ook was het onmogelijk om zonder veel omwegen een queue als doel van een webproxy te gebruiken. WCF gaat uit van een onafhankelijkheidsprincipe. De eigenlijke service kan geprogrammeerd worden zonder direct van de transportimplementatie uit te gaan. Pas door het toevoegen van bindings in de configuratie, zal de specifieke transporttechniek worden bepaald. Hierdoor kun je een service in iedere applicatie hosten en deze kan dan ook als service voor clients van verschillende oorsprong dienen door middel van multiple endpoints.

Configureren van multiple endpoints

Een WCF-service is niet gebonden aan één bepaald communicatieprotocol. Het is mogelijk dat een service op verscheidene protocollen zal luisteren. Zo zou een service niet alleen voor een client kunnen dienen die van een WCF-proxy gebruikmaakt, maar op

TransactionScopeRequired	Een Boolean die aangeeft of de method wel of niet in een transactie moet participeren. Als deze op true staat, zal de service meedoen in een transactie die de client heeft opgestart. Indien de client geen transactie meegeeft, zal WCF zelf impliciet een transactie opstarten waarin de method zal draaien.
TransactionAutoComplete	Een Boolean die aangeeft of de transactie automatisch als compleet beschouwd kan worden.. Als deze parameter op false staat, zal de transactie automatisch gecommit worden als er zich tijdens de uitvoering geen onbehandelde exception voordoet. Het is dus handig om de method generiek te maken en de uiteindelijke en totale commit door de client te laten triggeren.

Tabel 1.

```
[ServiceContract]
public interface IKlantenService
{
   [OperationContract]
   [TransactionFlow(TransactionFlowOption.Mandatory)]
   void Debet(String KlantID, double Bedrag);
   [OperationContract]
   [TransactionFlow(TransactionFlowOption.Mandatory)]
   void Credit(String KlantID, double Bedrag);
}

public class KlantenService : IKlantenService
{
   [OperationBehavior(TransactionAutoComplete = true, TransactionScope
Required = true)]
   public void Debet(String KlantID, double Bedrag)
   {
      //Implementatie
   }
   [OperationBehavior(TransactionAutoComplete = true, TransactionScope
Required = true)]
   public void Credit(String KlantID, double Bedrag)
   {
      //Implementatie
   }
}
```

Codevoorbeeld 6. Transacties in de service

het zelfde moment ook via een MSMQ-queue dezelfde messages kunnen inlezen. Dit kan eenvoudig worden gedaan door verscheidene instanties te maken van de servicehost en deze verschillende adressen te geven. Deze adressen kunnen elk van een verschillend protocol zijn. Voor alle protocollen dient een bijhorende binding aanwezig te zijn in de configuratiefile. Deze bindings bepalen uiteindelijk de mogelijke communicatiemechanismes. In codevoorbeeld 7 maakt een hostapplicatie twee endpoints beschikbaar. Door de bindingconfiguratie is bepaald hoe deze bereikbaar zijn. De methods in de klasse hebben geen idee op welke manier ze worden

```
internal static ServiceHost Host = null;
Host = new ServiceHost(typeof(KlantenService));
Host.Open();
```

Codevoorbeeld 7. Multiple endpoints

```
<services>
   <service
      name="KlantenLib.KlantenService"
      behaviorConfiguration="MyServiceTypeBehaviors">
      <endpoint
         address=" http://localhost:8081/Service/SendMessage"
               contract="KlantenLib.IKlantenService"
         binding="basicHttpBinding"/>
      <endpoint
         address="net.msmq://localhost/private/WCFQueue"
         binding="netMsmqBinding"
         bindingConfiguration="Binding1"
         contract="KlantenLib.IKlantenService"/>
   </service>
</services>
```

Codevoorbeeld 8. Configuratie bij multiple endpoints

```
[ServiceContract]
interface IKlantenService
{
   [OperationContract(IsOneWay = true)]
   void Operatie()
}
```

Codevoorbeeld 9. IsOneWay attribuut

```
[ServiceContract(SessionMode = SessionMode.Required,
   CallbackContract = typeof(IClientKlasse))]
public interface IServiceKlasse
{
   [OperationContract(IsOneWay = true)]
   void Vraag(string TheParam);
}
public interface IClientKlasse
{
   [OperationContract(IsOneWay = true)]
   void Antwoord(int location);
}
```

Codevoorbeeld 10. CallbackContract.

```
public class ServiceKlasse : IServiceKlasse
{
   private IClientKlasse callback = null;
   public ServiceKlasse()
   {
      callback = OperationContext.Current.GetCallbackChannel<IClientKlasse>();
   }
   public void Vraag(string TheParam)
   {
      //implementatie
      callback.Antwoord(100);
   }
}
```

Codevoorbeeld 11. Implementatie van de callback

aangesproken. Het is de host die de endpoints tot leven brengt en de binding die hun werking bepaalt.; zie codevoorbeeld 8.
Omdat MSMQ een asynchrone omgeving is, waardoor geen directe antwoorden via de queue verwacht kunnen worden, dient dit in het OperationContract aangegeven te worden via het attribuut IsOneWay; zie codevoorbeeld 9.

Duplex Communication-patroon

Door middel van het Duplex Communication-patroon kunnen clients een method in een service asynchroon aanroepen. Hierdoor zal de service antwoorden op de request door het uitvoeren van een method op de client die als callback is vastgelegd. De client publiceert zelf een WCF-method die door de service zal worden uitgevoerd met als resultaat parameters. Het patroon schrijft voor dat er naast de service-interface ook een interface gedefinieerd is hoe de callbackmethods er uit zien. De client dient deze interface te implementeren. Welke callbackinterface de aan te roepen service zal gebruiken, is via de CallBackContract-parameter in het ServiceContract te bepalen. Deze parameter moet een waarde krijgen die het type is van de callbackinterface. De callbackinterface is vastgelegd door het ServiceContract, waardoor de generatie van de clientproxy hiervan gebruik kan maken en code voor deze callback-klasse kan genereren tijdens het leggen van de WCF-servicereferentie of door middel van de svcutil-tool. De method in de service kan vanuit zijn context een referentie krijgen naar de klasse op de client door middel van OperationContext.Current.GetCallBackChannel(); zie codevoorbeeld 10 en 11.

Kurt Claeys (MCTS Biztalk 2006 en MCPD Enterprise Applications Developer) is werkzaam als .NET Architect bij Ordina Belgium en is betrokken bij de Microsoft Products Experts Unit met speciale aandacht voor Application Integration. Je kunt contact met hem opnemen via zijn blog www.devitect.net, kurt.claeys@ordina.be of blog.n-technologies.be.

Referenties

http://wcf.netfx3.com
http://msdn2.microsoft.com/en-us/netframework/aa663324.aspx
http://msdn2.microsoft.com/en-us/library//aa480190.aspx