Roland Guijt

WCF Callbacks

VEELZIJDIGHEID CALLBACK-FUNCTIONALITEIT VAN WCF

Een van de krachtige mechanismen geïntroduceerd in WCF is de callback-functionaliteit. Was het met .NET remoting nog een crime om het voor elkaar te krijgen, in WCF is het een verademing. Aan de hand van een voorbeeld gaat de auteur callbacks uitleggen, en alles wat daarbij komt kijken.

De crux van WCF-callbacks is: de server roept een methode aan op de client. De rollen client en server worden hierdoor tijdelijk omgedraaid. In het voorbeeld ga ik een typische toepassing hiervan uitwerken: Een chatsysteem. De chatclient is een winforms-applicatie waarmee je kunt aanmelden, afmelden en een bericht kunt sturen. Als er een bericht verstuurd wordt naar de server, dient de server er voor te zorgen dat alle aangemelde clients deze message ook te zien krijgen. Hiervoor maakt de server een callback.
De algemene werking van de applicatie is:

- Een client meldt zich aan.
- De client stuurt berichten en in de uiteindelijke applicatie krijgen alle aangemelde clients de berichten aangeboden via een callback.
- Een client meldt zich af.

Set-up

Voordat ik kan laten zien hoe je callbacks implementeert, heb ik eerst een uitgangssituatie nodig: een WCF-applicatie die het skelet vormt van de callback-implementatie. Misschien ook goed om te zien hoe het ook alweer zat met de basics van een WCF-applicatie. Tijdens het lezen van dit artikel, raad ik je aan om mee te kijken in de solution, gemaakt met Visual Studio 2008, die je kunt downloaden van www.oosterkamp.nl/WCFCallbacks.aspx. Een WCF-applicatie draait om een contract, dat een interface is die definieert wat de client aan methodes kan aanroepen op de server; zie codevoorbeeld 1.

Op de server is deze interface geïmplementeerd in een service-klasse. Deze klasse heb ik de ChatImplementor genoemd. Van deze klasse worden instanties gemaakt door de service. Het aantal instanties dat op de server wordt aangemaakt, is afhankelijk van de WCF-configuratie. Later in het artikel ga ik hier verder op in.
Ook is er een binding nodig die bepaalt hoe er gecommuniceerd wordt. In het set-upproject heb ik alles standaard gelaten. Er wordt dan een wsHTTPBinding gebruikt. Deze werkt hetzelfde als een HTTP-verbinding met een standaard webservice. Een service kan meerdere bindings hebben, die gedefinieerd worden door endpoints. Het laatste ingrediënt is het adres, in de vorm van een url waar de service te bereiken is. Naast het endpoint waarover de normale communicatie gaat verlopen, heb ik ook een metadata exchange (mex) endpoint gedefinieerd. Standaard laat WCF zijn interface namelijk niet aan de buitenwereld zien. Bij een mex endpoint gebeurt dit wel, volgens een afgesproken formaat. Dit formaat heet WSDL en is afkomstig van de originele webservice-technologie. De service host ik in een command line-applicatie die een app.config heeft, zoals te zien is in codevoorbeeld 2.
De app.config is in Visual Studio 2008 eenvoudig te maken of te veranderen door de WCF Service Configuration Editor; zie afbeelding 1. De client heeft ook een app.config waar iets soortgelijks in

Afbeelding 1. WCF Service Configuration Editor

```
[ServiceContract]
 public interface IChatContract
 {
    [OperationContract]
    void SendMessage(string Message);

    [OperationContract]
    void SignIn(string userName);

    [OperationContract]
    void SignOut(string userName);
  }
```

Codevoorbeeld 1. De chat interface

```
<service behaviorConfiguration="Oosterkamp.ChatService.ChatServiceBehavior"
name="Oosterkamp.ChatService.ChatImplementor">
   endpoint address="" binding="wsHttpBinding" contract="Oosterkamp.
ChatService.IChatContract">
      <identity>
         <dns value="localhost" />
      </identity>
   </endpoint>
   <endpoint address="mex" binding="mexHttpBinding"
 contract="IMetadataExchange" />
   <host>
      <baseAddresses>
         <add baseAddress="http://localhost:8731/ChatService/" />
      </baseAddresses>
   </host>
</service>
</services>
```

Codevoorbeeld 2. App.config van de host-applicatie

```
public partial class ClientMainForm : Form
{
  private ChatContractClient chatClient;

  public ClientMainForm()
  {
    InitializeComponent();
    chatClient = new ChatContractClient();
    chatClient.Open();
  }

  protected override void OnClosed(EventArgs e)
  {
    base.OnClosed(e);
    chatClient.Close();
  }

  private void SendButton_Click(object sender, EventArgs e)
  {
    chatClient.SendMessage(messageTextBox.Text);
  }
}
```

Codevoorbeeld 3. Client-implementatie

staat en dat een winforms-applicatie is. Deze heeft een proxy-klasse die gegenereerd is op basis van de metadata in de server, de WSDL. Dit heet een service reference. Alle methode-aanroepen lopen via deze proxy-klasse. Met een paar regels code kun je zodoende via het proxy-object tegen de server praten, zoals je kunt zien in codevoorbeeld 3.

Je kunt nu een bericht invoeren op de client, dat dan naar de server wordt gestuurd. Een aardig begin, maar er zijn een paar handelingen nodig voordat het gaat werken als een chatsysteem:

- InstanceContextMode aanpassen: Hiermee bepaal je op welke manier serviceobjecten worden gecreëerd op de server.
- Duplex binding aanpassen: De binding moet gewijzigd worden, zodat duplex-verkeer mogelijk wordt.
- SessionMode aanpassen: Hiermee bepaal je of en op welke manier er met sessies wordt omgegaan op de server.
- Callback-contract maken: Definitie van het contract dat geldt tussen server en client.
- Concurrency-mode eventueel aanpassen.
- Overige codeaanpassingen op zowel server als client.

InstanceContextMode

Aan de serverkant kun je bepalen op welke manier er instanties van de serverklasse (ChatImplementor) worden geïnstantieerd. In tabel 1 staat een opsomming van de drie mogelijke instellingen. De manier waarop je dit doet, is afhankelijk van het soort applicatie. Als je bijvoorbeeld een applicatie maakt die het weer teruggeeft, is PerCall wellicht de juiste oplossing. PerSession is bruikbaar als je

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.Single)]
public class ChatImplementor : IChatContract
```

Codevoorbeeld 4. InstanceContextMode configureren

```
<endpoint address="" binding="wsDualHttpBinding" contract="Oosterkamp.
ChatService.IChatContract">
```

Codevoorbeeld 5. De binding configureren

een applicatie maakt waarin je naar aanleiding van eerdere opgaaf van data functionaliteit uitvoert, bijvoorbeeld een service waar je hypotheekberekeningen kunt doen. De chatserver moet een lijst bijhouden welke clients er zijn. De bedoeling is dat bij het ontvangen van een bericht, de server door de lijst itereert en alle clients het bericht stuurt door middel van een callback. Er is slechts één lijst voor alle clients. Daarom is in deze applicatie gekozen voor Single als InstanceContextMode. Het instellen doe je met een attribute boven de klasse die het servicecontract implementeert; zie codevoorbeeld 4.

Duplex binding

Het tweede dat aangepast moet worden is de binding. De standaard wsHTTPBinding is alleen geschikt voor éénrichtingsverkeer. Dat betekent dat callbacks niet kunnen werken, omdat er tegelijkertijd heen en weer gecommuniceerd moet worden. In dit artikel maak ik gebruik van een wsDualHTTPBinding, maar er zijn meer "duplex" bindings:

- NetTCPBinding
- NetNamedPipeBinding
- NetPeerTCPBinding
- NetTCPConextBinding

Het mooie van WCF is dat je dit met slechts één aanpassing in de app.config op zowel de host als de client kunt instellen; zie codevoorbeeld 5. Address is hier leeg, omdat het standaard adres gebruikt wordt, ook gedefinieerd in de app.config: http://localhost:8731/ChatService/

SessionMode

Als onze client meerdere calls doet naar de server, dan moet de server weten dat deze calls bij dezelfde client horen. Dit heet session in WCF. De session in deze applicatie is analoog aan een ingelogde client. Je zou dit voor onze chat-applicatie kunnen oplossen om bij elke aanroep naar SendMessage een gebruikersnaam mee te sturen. Maar de functionaliteit zit al in WCF en kun je met een attribute-parameter aanzetten. Zoals te zien in codevoorbeeld 6. In tabel 2 staan de mogelijke waardes van de SessionMode-enumeratie.

Afbeelding 2. Het eerste client-scherm

Naam	Omschrijving	Voordelen	Nadelen	.NET remoting equivalent
PerCall	Per aanroep naar de server wordt een object gecreëerd. In ons voorbeeld voor elke sendmessage-aanroep een nieuw object. De service is hiermee stateless.	Geen concurrency-problemen. Als er geen serveractiviteit is, staan er geen onnodige objecten in het geheugen	Er is geen mogelijkheid iets te onthouden in het serverobject (state).	Single call
PerSession	Voor iedere client is er een serveobject aanwezig.	Per client is het mogelijk om state te bewaren in het serverobject.	De schaalbaarheid is slechter: bij veel clients worden veel resources gebruikt op de server. Geen mogelijkheid om globaal state te bewaren in het serverobject.	Client activated object
Single	Voor alle calls en clients samen is er maar één serverobject.	Globaal state bewaren.	De kans op concurrency-problemen is groot.	Singleton

Tabel 1. InstanceContextMode-opties

```
[ServiceContract(SessionMode = SessionMode.Required)]
```
Codevoorbeeld 6. Session configureren

```
public interface IChatClientContract
{
   [OperationContract(IsOneWay = true)]
   void MessageReceived(string Message);
}
```
Codevoorbeeld 7. Het callback contract

```
[ServiceContract(CallbackContract = typeof(IChatClientContract),
SessionMode = SessionMode.Required)]
public interface IChatContract
```
Codevoorbeeld 8. Aangeven welk callback-contract er gebruikt moet worden

Callback-contract

Bij callbacks doet de server een methodeaanroep bij de client. Er is een apart contract nodig dat gedefinieerd is aan de serverkant en geïmplementeerd aan de client-kant. Dit contract staat gedefinieerd in dezelfde file als waar de proxy-klasse in staat. Hij wordt gegenereerd op het moment dat er een metadata-exchange plaatsvindt; zie codevoorbeeld 7. De parameter IsOneWay bij het attribute OperationContract van de methode MessageReceived maakt de aanroep naar deze method een 'fire and forget'-actie. Er wordt door de server niet gewacht tot er een bevestiging van de client komt dat alles goed is verlopen. Dit zou een deadlock kunnen veroorzaken, bijvoorbeeld als de client wacht tot de SendMessage-aanroep klaar is en de server wacht op de MessageReceived-aanroep. De serverconfiguratie is verrassend eenvoudig. Er hoeft alleen bij het servicecontract aangegeven te worden welk callback-contract gebruikt moet worden; zie codevoorbeeld 8. Deze attribute-parameter is het enige wat je hoeft te doen om te zorgen dat bij een volgende metadata-exchange de interface bij de client bekend is.

Client-implementatie

De service-reference moet nu vernieuwd worden, omdat ik nu het callback-contract heb ingesteld op de server. Na het updaten van de service-reference kun je op de client de klasse maken die de interface implementeert. Merk op dat deze niet dezelfde naam heeft als het origineel op de server. Dit komt omdat de interface geconstrueerd is op basis van de metadata die afkomstig zijn van de server; zie codevoorbeeld 9.

Het enige dat ik doe is een event afvuren met het bericht in de EventArgs. De client-code moet nu aangepast worden, zodat een instantie van deze klasse geregistreerd wordt als callback-object. De InstanceContext wordt normaal impliciet gecreëerd als je de proxy instantieert. Echter, bij callbacks moet je nog vertellen wat het callback-object moet zijn. Dit doe je door het callback-object mee te geven in de constructor van InstanceContext en het InstanceContext-object weer mee te geven bij het creëeren van de proxy; zie codevoorbeeld 10.

InstanceContext is een object dat alle informatie bevat over de context van de service-instantie. Je kunt er bijvoorbeeld de channels mee configureren als je niet gebruikmaakt van de declaratieve manier van configureren in de app.config. Er zit ook een event in dat afgaat als er iets fout gaat (Faulted). De methode callBackImplementor_MessageIncoming voegt het bericht toe aan een multiline TextBox die ik op het formulier heb gezet.

Naam	Omschrijving
Allowed	Als de binding session ondersteunt, wordt het gebruikt, anders niet.
Not allowed	Session wordt niet gebruikt.
Required	Binding moet session ondersteunen, anders treedt er een exceptie op.

Tabel 2. SessionMode opties

```
class ChatClientImplementor : Oosterkamp.ChatService.Client.
IChatContractCallback
{
   public event EventHandler<MessageEventArgs> MessageIncoming;

   public void MessageReceived(string Message)
   {
      if (MessageIncoming != null)
         MessageIncoming(this, new MessageEventArgs(Message));
}
```
Codevoorbeeld 9. De implementatie van de geconstrueerde interface

```
public ClientMainForm()
{
   InitializeComponent();
   ChatClientImplementor callBackImplementor =
      new ChatClientImplementor();
   callBackImplementor.MessageIncoming += new EventHandler
      <MessageEventArgs>(callBackImplementor_MessageIncoming);
   InstanceContext context = new InstanceContext(callBackImplementor);
   chatClient = new ChatContractClient(context);
   chatClient.Open();
}
```
Codevoorbeeld 10. Meegeven van een InstanceContext aan de proxy

Server-implementatie

Van de server is in deze configuratie maar één object aanwezig voor alle clients. Dit betekent dat ik een lijst van alle ingelogde clients kan bijhouden. Er is hiervoor een klasse gecreëerd die een client voorstelt, genaamd ChatUser. Het relevante stuk voor callbacks is afgedrukt; zie codevoorbeeld 11.

Het object OperationContext bevat informatie over een bepaalde client. Je kunt bijvoorbeeld kijken of de client nog actief is en security-zaken regelen. Door middel van OperationContext kun je een referentie naar het client-object krijgen. Dit is een object dat IChatClientContract implementeert. Hiermee kan de methode MessageReceived aangeroepen worden op de client (de callback). Met de generic methode GetCallBackChannel krijg je een proxy-object terug waar je de aanroep kunt doen voor een specifieke client. De OperationContext kun je eenvoudig opvragen door de statische property Current te gebruiken op de OperationContext-klasse. Op het moment dat een nieuwe client inlogt, wordt een nieuw ChatUser-object in de lijst geplaatst met dit object; zie codevoorbeeld 12. De lijst van clients (chatUsers) is een generic List van ChatUsers,

```
internal IChatClientContract CallBackChannel
{
   get
   {
      if (context != null)
         return context.GetCallbackChannel<IChatClientContract>();
      else
         return null;
   }
}
private OperationContext context;
internal OperationContext Context
{
   get
   {
      return context;
   }
   set
   {
      context = value;
   }
}
```
Codevoorbeeld 11. Het relevante stuk van de ChatUser klasse

```
public void SignIn(string userName)
{
   ChatUser newUser = new ChatUser(userName);
   newUser.Context = OperationContext.Current;
   chatUsers.Add(newUser);
}
```

Codevoorbeeld 12. Opslaan van de ingelogde gebruiker

```
foreach (ChatUser user in chatUsers)
{
   //Controleer of de client nog actief is
   if (user.Context.Channel.State == CommunicationState.Opened)
      user.CallBackChannel.MessageReceived(message);
   else
      SignOut(user);
}
```

Codevoorbeeld 13. Controleren of de client nog actief is

```
[ServiceBehavior(InstanceContextMode = InstanceContextMode.Single,
ConcurrencyMode = ConcurrencyMode.Single)]
```

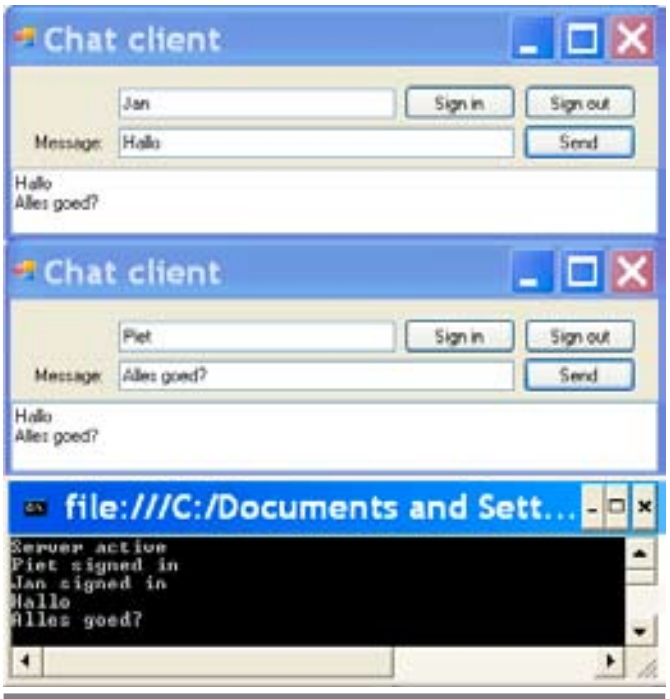
Codevoorbeeld 14. De ConcurrencyMode property in het ServiceBehavior attribute

die ik doorloop met een foreach op het moment dat een nieuw bericht wordt ontvangen; zie codevoorbeeld 13.
Er wordt gekeken of de client er nog wel is. Zo niet, dan wordt de client uit de lijst verwijderd. De criteria waarmee WCF bepaalt of de client nog aanwezig is, zijn te configureren. Dit kun je bijvoorbeeld doen door een bindingconfiguration te maken, waarin je een time-out specificeert en deze aan je binding koppelt.

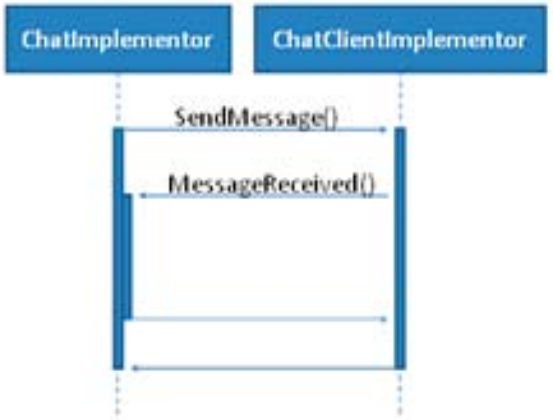
ConcurrencyMode

De stappen die in de voorbeeldapplicatie worden doorlopen, schematisch weergegeven in afbeelding 4, zien er zo uit:

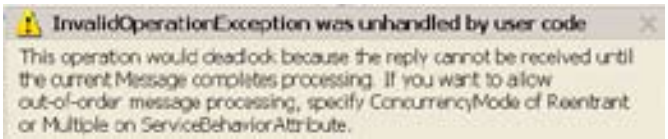
- De client doet een aanroep naar de server (SendMessage).
- De client wacht totdat die aanroep is uitgevoerd.
- Terwijl de client wacht, vindt een callback plaats van de server, die ook door de client wordt afgehandeld in een andere thread.
- De client krijgt een bevestiging van de server dat de SendMessage-aanroep succesvol is uitgevoerd.



Afbeelding 3. Eindresultaat



Afbeelding 4. De applicatie in schema



Afbeelding 5. De exception die je krijgt als de callback niet OneWay is bij een ConcurrencyMode van ConcurrencyMode.Single

Er is hier sprake van concurrency. Dat wil zeggen: er worden zaken tegelijkertijd afgehandeld door server en client.
Ook concurrency is in WCF te regelen met het ServiceBehavior-attribute; zie codevoorbeeld 14. De ConcurrencyMode-enumeratie staat beschreven in de tabel 3. Onze chat-applicatie werkt, ondanks dat de concurrencymode Single is. Dit komt omdat de callback-method OneWay is. De server geeft daarbij opdracht voor het uitvoeren van de methode en wacht verder niet op een bevestiging of de bewerking is geslaagd. Als je de IsOneWay-attributeparameter in de IchatClientContract-interface verwijdert, krijg je een build-error, zoals te zien in afbeelding 5.

Naam	Omschrijving
Single	Het serverobject wordt gelockt totdat de aanroep klaar is.
Reentrant	Als single, maar de server mag callbacks maken naar de client
Multiple	De server en de client mogen zonder beperkingen calls naar elkaar uitvoeren.

Tabel 3. Concurrency modes

Eenvoudig configureerbaar

WCF-callbacks zijn niet alleen veelzijdig, je kunt ze ook eenvoudig configureren. Het meeste configuratiewerk zit in het gebruiken van attributes. De veelzijdigheid uit zich in het feit dat je vrij bent om sessies, serverobjecten, bindings en concurrencymode te configureren naar eigen wens, wat in het verlengde ligt van het WCF-concept.

Roland Guijt is trainer bij Oosterkamp training|consultancy (www.oosterkamp.nl), een organisatie gespecialiseerd in .NET. Op dit gebied geeft Roland al jaren enthousiast training, zowel op open inschrijving, als op maatwerkbasis. Om vanuit de praktijk de theorie goed over te brengen, wisselt hij het geven van trainingen af met consultancy, zoals recentelijk een project met WCF. Zijn blog tref je aan op http://blogs.oosterkamp.nl/blogs/roland/.