# WCF & WF in het .NET Framework 4

## Uitdagingen op een elegante manier oplossen

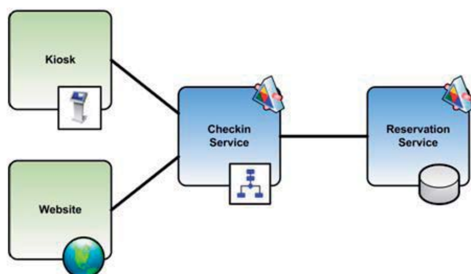
Willem Meints, Alexander Molenkamp, Edwin van Wijk

In maart 2010 komt versie 4 versie van het .NET Framework uit. In deze versie is er onder andere op het gebied van Windows Communication Foundation (WCF) en Workflow Foundation (WF) veel nieuws te melden. Aan de WCF kant is vooral veel nieuwe functionaliteit toegevoegd, zoals een vereenvoudigd configuratiemodel, ondersteuning van het WS-Discovery protocol en Routing Services. WF is van de grond af opnieuw opgebouwd.

Ook het raakvlak tussen WF en WCF is grondig aangepakt. Nieuwe Messaging activities en uitgebreidere Correlation mogelijkheden in WF4 maken het eenvoudiger om middels WCF met een workflow te communiceren en vanuit een workflow met WCF services te communiceren. Een nieuwe feature waarbij de sterke punten van WCF en WF samenkomen is Declarative Services, waarmee het eenvoudiger wordt een WCF service te implementeren op basis van een WF workflow. In dit artikel zullen we een aantal wijzigingen en uitbreidingen beschrijven en demonstreren aan de hand van een eenvoudig voorbeeldscenario. Het voorbeeldscenario waarmee we zaken willen demonstreren is het check-in proces van een fictieve luchtvaartmaatschappij 'OneWay Air'. Deze maatschappij biedt passagiers twee verschillende mogelijkheden om in te checken. Passagiers kunnen maximaal 24 uur voor vertrektijd inchecken via de OneWay Air website of via kiosks op het vliegveld. Het checkin-process bestaat uit drie stappen. Allereerst zoekt de passagier zijn reservering op in het systeem met behulp van zijn e-ticket nummer. Vervolgens is er de mogelijkheid om de vooraf toegewezen stoel te wijzigen. Dit kan meerdere keren totdat de stoelkeuze door de passagier wordt bevestigd. Als dat is gebeurd wordt de boarding pass afgedrukt.

### Voorbeeldoplossing

Voor dit scenario hebben we een voorbeeldoplossing gemaakt op basis van de Bèta 2 versies van het .NET Framework 4 en Visual Studio 2010. In Figuur 1 is een schematisch overzicht van deze voorbeeldoplossing te zien.




FIGUUR 1: VOORBEELDSCENARIO ARCHITECTUUR

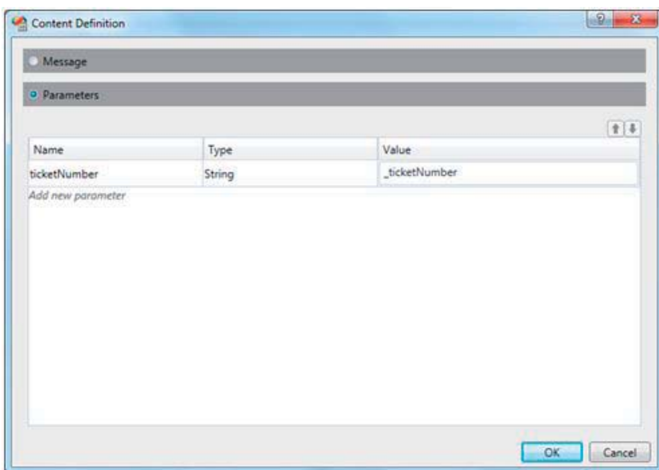
De algemene logica voor het checkin process wordt aangeboden door de CheckIn service. Deze service implementeren we middels een workflow. De website en de kiosk applicatie maken beiden gebruik van de CheckIn service. De CheckIn service maakt op zijn beurt gebruik van de Reservation service voor het ophalen en opslaan van klant- en reserveringsgegevens. Voor de implementatie van de CheckIn workflow maken we gebruik van de nieuwe Flowchart activity die het mogelijk maakt om workflows als flowcharts te modelleren. Flowcharts sluiten beter aan op de manier waarop veel ontwikkelaars nadenken over processen en maken het in tegenstelling tot sequential workflows mogelijk om eenvoudig terug te keren naar vorige stappen. Flowcharts kunnen ook gebruikt worden als onderdeel van een sequence en vice-versa. In de vorige versie van WF bestond deze vorm van workflow nog niet en moest men gebruik maken van een state machine workflow om hetzelfde effect te verkrijgen. WF 4 biedt in principe geen ondersteuning meer voor state machine workflow maar de functionaliteit van een state machine is te simuleren met behulp van de nieuwe Flowchart activity in combinatie met een Pick activity (hier komen we later op terug).

De CheckIn service in onze voorbeeldoplossing wordt gestart zodra er een StartCheckin bericht ontvangen wordt. Dit bericht bevat het e-ticket nummer van de passagier waarmee de workflow bij de Reservation service de reserveringsgegevens kan ophalen. De opgehaalde reserveringsgegevens worden vastgehouden in de workflow en geretourneerd aan de client applicatie. Vervolgens gaat de workflow wachten op een stoelwijzigings- of bevestigingsbericht met behulp van de Pick activity. De Pick activity lijkt op de Listen activity uit WF3.x; er kan een aantal verschillende paden gedefinieerd worden waarbij elk pad een trigger/event en een actie heeft. Wanneer een trigger/event voor een bepaald pad binnenkomt, zal de bijbehorende actie bij dit pad worden uitgevoerd en zullen eventuele overige paden automatisch worden geannuleerd. In onze service is er een

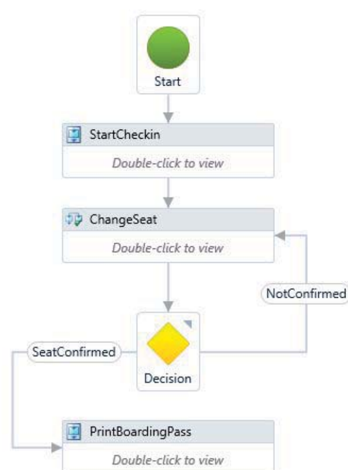
pad dat reageert op de ontvangst van een stoelwijzigingsbericht en een pad dat reageert op de ontvangst van een stoelbevestigingsbericht. We hebben op die manier dus een statemachine met 2 states gesimuleerd middels een flowchart en een pick activity met 2 paden. Bij binnenkomst van een stoelbevestigingsbericht wordt de status van de reservering gewijzigd naar SeatConfirmed. Na de Pick activity wordt de status van de reservering in een FlowDecision activity getest op SeatConfirmed en indien deze test faalt gaat de workflow terug naar de Pick activity. Het gebruik van de nieuwe Flowchart activity maakt het proces inzichtelijker aangezien de herhaling nu minder verstopt is. De CheckIn service zal na een bevestigingsbericht de reserveringsstatus wijzigen naar CheckedIn. De laatste stap is het afdrukken van de boarding passes. De CheckIn service levert de gegevens aan als antwoord op de PrintBoardingPass operatie. De client applicaties zijn vervolgens verantwoordelijk voor het daadwerkelijk afdrukken op papier. De CheckIn service workflow is te zien in Figuur 3. We hebben nu de workflow in onze voorbeeldoplossing in detail beschreven. We zullen in de rest van dit artikel kijken naar hoe vanuit de 'buitenwereld' met de workflow wordt gecommuniceerd.

## Declarative Services

De workflow waarmee we de CheckIn service implementeren moet als webservice aangeroepen kunnen worden vanuit de client applicaties. We maken hiervoor gebruik van de Declarative Services feature. Een declarative service is een workflow service die volledig in XAML wordt beschreven. Het gebruik van Declarative




**FIGUUR 2: RECEIVEACTIVITY ARGUMENTEN DIALOG**




**FIGUUR 3: CHECKIN SERVICE WORKFLOW**

services biedt een extra abstractielaag hetgeen inhoudt dat beschreven wordt wat de service moet doen. Dit in tegenstelling tot het schrijven van imperatieve code (zoals C# of VB.NET) waarbij je beschrijft hoe het werk gedaan moet worden. Omdat de service wordt geïmplementeerd als workflow, profiteert deze tevens van de voordelen die de WF runtime biedt, zoals bijvoorbeeld tracking en persistence.

Als we naar het Visual Studio project voor de CheckIn service kijken, zien we dat dit bestaat uit een bestand met de extensie '.xamlx' (de declarative workflowservice definitie) en een web.config bestand. Met het .NET Framework 4 is het mogelijk om .xamlx bestanden te hosten in IIS (dus er is geen additioneel .svc bestand nodig) of middels de WorkflowServiceHost (self hosted). De inhoud van de web.config is zeer summier dankzij het nieuwe configuratiemodel voor WCF

```
<configuration>
  <system.web>
    <compilation debug="true" targetFramework="4.0" />
  </system.web>
  <system.serviceModel>
    <behaviors>
      <serviceBehaviors>
        <behavior>
          <serviceMetadata httpGetEnabled="true" />
          <serviceDebug includeExceptionDetailInFaults="true" />
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

Als er geen specifieke WCF configuratie is opgegeven voor een service, zal de WCF runtime de service automatisch voorzien van een standaard endpoint en een aantal behaviors. In de web.config hebben we de servicebehaviors expliciet gespecificeerd. Omdat deze configuratie geen naam heeft, wordt deze als default gebruikt en zal onze CheckIn service voorzien worden van de gespecificeerde servicebehaviors.

## WCF Messaging Activities

WF 4 bevat een viertal WCF messaging activities. Deze zijn beschreven in Tabel 1.

Bij een Receive activity kan middels de ReceiveContent property worden gespecificeerd wat de input data voor de operatie is. Er kan uit twee typen content worden gekozen: message content en parameter content. Als de input een Message object of een messagecontract type is, dan moet de input als message content worden gespecificeerd. Is dat niet het geval (bijvoorbeeld bij simple types of data-contract types), dan mag hier gekozen worden tussen message- en parameter content. Het voordeel van parameter content is dat er meerdere individuele parameters kunnen worden opgegeven. De waarde van een ontvangen message of parameter kan worden toegekend aan een variabele (zie Figuur 2). Op basis van de Messaging activities die zijn gebruikt in de workflow, wordt de metadata voor de service automatisch gegenereerd. Als de service is uitgerust met een MetaData behavior kan de WSDL door clients worden opgehaald en gebruikt worden om een proxy te genereren. De gegenereerde WSDL kan worden beïnvloed met een aantal eigenschappen van de messaging activities. Zo is het niet alleen mogelijk om bij een Receive activity de operatienaam te specificeren, maar kan ook expliciet worden gespecificeerd welke XML namespace er moet worden gebruikt en wat de naam van het servicecontract is. Tevens

| Toolbox item | Omschrijving |
|---|---|
| Send | Deze activity kan worden gebruikt om een bericht naar een externe webservice te verzenden . |
| Receive | Deze activity kan worden gebruikt om een bericht te ontvangen van een client. |
| ReceiveAndSendReply | Dit is een combinatie van de Send en Receive activities. Deze activity combinatie kan worden gebruikt om een bericht te ontvangen en hierop later op te antwoorden met een bericht. |
| SendAndReceiveReply | Dit is net als ReceiveAndSendReply een combinatie van de Send en Receive activities. Deze combinatie kan worden gebruikt om een bericht te verzenden en om vervolgens te wachten op een antwoord van de service waar het bericht naar is verzonden. |

**TABEL 1**

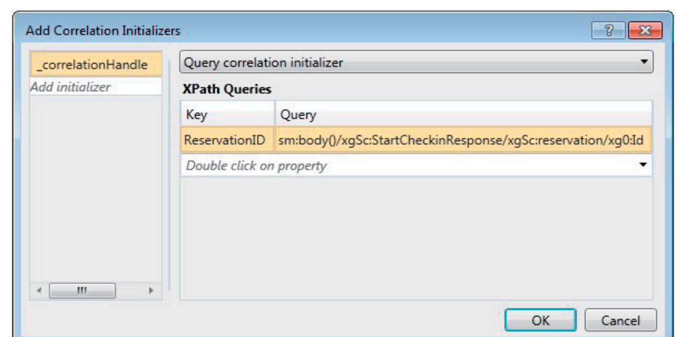
is het mogelijk om zowel de SOAP action als de SOAP reply-action op te geven als daar behoefte aan is. Door de flexibiliteit waarmee de Messaging activities kunnen worden ingezet in de workflow, wordt een groot aantal verschillende communicatiescenario's ondersteund. Zo kan er bijvoorbeeld eenvoudig een one-way operatie worden gedefinieerd door alleen een Receive activity zonder een corresponderende SendReply op te nemen. Wat opvalt aan de manier waarop de Messaging activities zijn geïmplementeerd is dat er geen gebruik wordt gemaakt van contract-first design. Het is niet mogelijk om op basis van een vooraf samengestelde WSDL en schema een set Receive en SendReply activities te genereren. Dit hoeft echter geen belemmering te zijn voor ontwikkelaars om workflow services te implementeren die voldoen aan een eerder opgesteld contract en schema, mede doordat er zoveel mogelijkheden zijn om de Messaging activities te configureren. Er schuilt wel een risico in het feit dat een verandering in de messaging activities in een workflow ook een verandering in het contract betekent. Hier moet een ontwikkelaar zich goed van bewust zijn.

Er zijn twee manieren waarop vanuit een workflow met webservices kan worden gecommuniceerd: het zetten van een servicereference naar de service of gebruikmaken van de WCF Messaging activities. We hebben er in de CheckIn service van ons voorbeeldscenario voor gekozen om een servicereference naar de Reservation service toe te voegen. Visual Studio 2010 genereert op basis van deze servicereference automatisch een activity uit per serviceoperatie en plaatst deze in de toolbox van Visual Studio. Deze activities kunnen we vervolgens in de workflow gebruiken om de Reservation service aan te roepen. Dit is in veel gevallen de meest eenvoudige manier om met een andere webservice te communiceren.

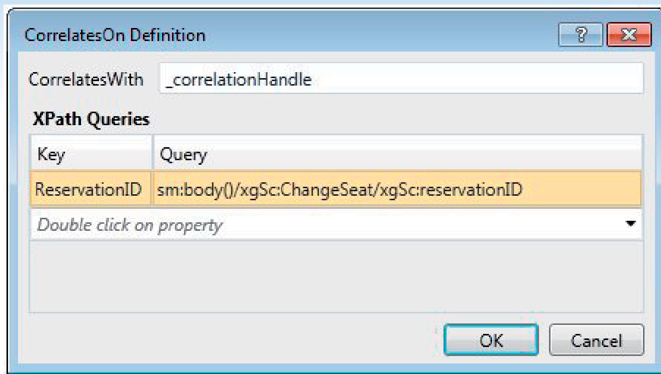
## Message Correlation

We hebben gezien dat het check-in process in ons voorbeeldscenario bestaat uit meerdere stappen. Zowel het starten van het check-in proces, als het wijzigen en bevestigen van de stoel zijn acties die worden gestart door de clients. Om ervoor te zorgen dat alle acties voor één reservering op dezelfde workflow uitkomen, is het noodzakelijk om een vorm van correlatie te gebruiken. In WF 3.5 was correlatie alleen mogelijk op basis van een zogenaamde message context. Dit betekende dat één van de WCF context bindings gebruikt moest worden, zoals BasicHttpContextBinding. Het workflow instance ID werd bij een aanroep van de workflow geretourneerd aan de client en het was de taak van de client om dit ID met verdere berichten weer mee te geven. Dit heeft een aantal nadelen waaronder het feit dat de client op de hoogte moet zijn van dit protocol en zelf het workflow instance ID moet bijhouden. Daarnaast was het niet mogelijk om een workflow te starten met een one-way operatie, omdat de benodigde context hierbij niet geretourneerd kan worden aan de client. WF 4 ondersteunt ook context-based correlation, maar introduceert daarnaast een aantal nieuwe varianten van message correlation, waaronder Content-based correlation. Dit is de meest uitgebreide vorm van correlatie in WF 4 waarbij meerdere verzoeken naar een workflow aan elkaar gecorreleerd worden op basis van gegevens uit het bericht. In de CheckIn service in onze voorbeeldoplossing maken we gebruik van een content-based correlatie die wordt gestart wanneer de client de StartCheckin operatie aanroept. De eerste stap voor het opzetten van de correlatie is het kunnen starten van de workflow op basis van een Receive activity. Hiervoor dient de CanCreateInstance property van de eerste Receive activity die wordt aangeroepen op true worden gezet. Deze creëert bij een aanroep de workflow instantie. De volgende stap is het bepalen welke gegevens uit het bericht moeten worden gebruikt om een unieke sleutel te vormen waarop de workflow instantie kan worden teruggevonden. Hiervoor gebruiken we XPath query's die gegevens uit het bericht opleveren. Een correlatie wordt geïnitialiseerd door een CorrelationInitializer. Op welk moment we de correlatie initialiseren is afhankelijk van wie de identificerende gegevens op basis waarvan zal worden gecorreleerd aanlevert. Als de client deze aanlevert kan de correlatie worden geïnitialiseerd bij het ontvangen van het eerste bericht. Dit doen we door een CorrelationInitializer te koppelen aan de Receive activity. Als de identificerende gegevens niet door de client worden aangeleverd maar in de service worden bepaald, moet de correlatie bij het retourneren van deze gegevens aan de client worden geïnitialiseerd. Dit doen we door een CorrelationInitializer aan de Send of SendReply activity te koppelen. In onze voorbeeldoplossing willen we correleren op basis van het reserveringsnummer. Het initiële bericht dat de client stuurt bevat echter alleen het e-ticketnummer. Het reserveringsnummer wordt aan de hand van dit e-ticketnummer door de service opgezocht. We moeten de SendReply activity dus voorzien van een CorrelationInitializer. Het zetten van de CorrelationInitializer property van het activity gebeurt met behulp van een dialoogvenster als in Figuur 4 wordt getoond.

Een correlatie wordt altijd gekoppeld aan een CorrelationHandle. We hebben hiervoor een variabele genaamd _correlationHandle van het type CorrelationHandle in de workflow aangemaakt. Voor elke CorrelationHandle aan de linkerzijde kan aan de rechterzijde één van de eerder beschreven correlation typen worden gekozen. We kiezen hier voor Query based correlation. Nadat hiervoor is gekozen wordt de lijst rechts onderin het scherm ingeschakeld en moeten er één of meerdere XPath expressies worden opgegeven waarmee wordt aangegeven waar de identificerende gegevens zich in het bericht bevinden. Het dialoogvenster helpt bij het maken van de XPath expressies door de lijst te tonen met argumenten (parameters of een message) die voor het activity zijn gespecificeerd. We hebben in onze voorbeeldoplossing het reservationId dat wordt geretourneerd aan de client als identificerend gegeven voor de correla-




**FIGUUR 4: CORRELATION INITIALIZER DIALOG**




FIGUUR 5: 'CORRELATES ON' DIALOG

tie gespecificeerd (zie Figuur 4). Als het check-in proces van onze voorbeeldoplossing is gestart, kan door middel van de ChangeSeat operatie de stoel voor een reservering worden gewijzigd. We willen op basis van het reservationId correleren aan de eerder gestarte workflow. Op de Receive activity voor de ChangeSeat operatie hebben we de CorrelatesWith property gezet met de _correlationHandle variabele die we in de correlation initializer hebben geïnitialiseerd. Bij de CorrelatesOn property van de Receive activity hebben we een query opgegeven die het reservationId in het bericht voor de stoelwijziging oplevert (zie Figuur 5).

Op dezelfde wijze hebben we de correlatie geconfigureerd voor de ConfirmSeat en PrintBoardingPass operaties. Het voordeel van content-based correlatie ten opzichte van context-based correlatie is dat de client niets hoeft te doen met workflow identifiers om toch op dezelfde instantie van een eerder gestarte workflow uit te komen. In plaats hiervan kan er met zinvolle bedrijfsgegevens worden gewerkt om op dezelfde instantie van de workflow uit te komen. Dit maakt het gebruik van de CheckIn service transparanter voor ontwikkelaars en is voor clients niet zichtbaar dat de serviceimplementatie een workflow is.

## WS-Discovery

Onze voorbeeldoplossing biedt nu alle benodigde functionaliteit om passagiers te laten inchecken. Als we onze oplossing nu zouden willen opschalen door bijvoorbeeld onze services naar andere (snellere) machines te verplaatsen, dan zou dat betekenen dat o.a. dat alle clients moeten worden aangepast om deze te voorzien van het nieuwe adres van de CheckIn service. Dit brengt veel extra werk met zich mee omdat bijvoorbeeld de kiosks fysiek moeten worden bezocht om een dergelijke wijziging door te voeren. Om in de toekomst kosten te besparen, is besloten het systeem zodanig aan te passen dat het verhuizen van services geen impact op de clients heeft.

Om clients onafhankelijk van de locatie van een service te maken is er een mechanisme nodig om de daadwerkelijke locatie van een service at runtime te kunnen achterhalen. Een manier om dit te doen is gebruik te maken van de WS-Discovery standaard. Deze OASIS standaard beschrijft een op SOAP gebaseerd protocol waarbij een client middels een broadcast op het netwerk kan vragen waar een bepaalde service leeft. De desbetreffende service kan dan antwoorden waarbij hij zijn locatie teruggeeft. De client kan vervolgens de service op deze locatie aanroepen. Services kunnen ook middels een broadcast aangeven dat ze online komen of offline gaan. Dit is een zeer vereenvoudigde beschrijving van de WS-Discovery standaard, maar dat is voldoende informatie voor dit artikel. Voor een meer gedetailleerde beschrijving wordt verwezen naar de specificatie van de standaard. In WCF 4 is een WS-Discovery implementatie beschikbaar. De functionaliteit van WS-Discovery wordt geleverd door de classes in Tabel 2 (ze zijn allemaal te vinden in de namespace 'System.ServiceModel.Discovery').

Om Discovery te demonstreren in onze voorbeeldoplossing gaan we er voor het gemak van uit dat de clients en services allemaal binnen een enkel netwerksegment leven (hier komen we later op terug). We rusten de CheckIn service uit met het DiscoveryBehavior en een UdpDiscoveryEndpoint (zie Tabel 2). Dit doen we in de web.config. Eerder zagen we dat we gebruikmaakten van de stanaardconfiguratie feature en de web.config van de CheckIn service nagenoeg leeg was. Omdat we de service nu willen uitrusten met behaviors, hebben we een specifieke configuratie gemaakt voor de CheckIn service.

```
<configuration>
  <system.web>
    <compilation debug="true" targetFramework="4.0" />
  </system.web>
  <system.serviceModel>
    <services>
      <service
        behaviorConfiguration="CheckInServiceBehaviors"
        name="CheckInService">
        <endpoint
          name="CheckInService"
        address="http://localhost:9000/airlinesample/services/
        checkin/implementation.xamlx"
          binding="basicHttpBinding"
          contract="ICheckInService" />
        <endpoint name="UDPDiscovery" kind="udpDiscoveryEndpoint" />
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
        <behavior name="CheckInServiceBehaviors">
          <serviceMetadata httpGetEnabled="true" />
          <serviceDebug includeExceptionDetailInFaults="true" />
          <serviceDiscovery/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

We zien dat er een endpoint van het type 'udpDiscoveryEndpoint' wordt toegevoegd aan de service. Deze manier van het specificeren van een endpoint (m.b.v. een endpoint-'kind') is nieuw in WCF 4. Er zijn een aantal standaard endpoints gedefinieerd waaronder een udpDiscoveryEndpoint. Daarnaast wordt er in de configuratie een 'ServiceDiscovery' behavior toegevoegd. Nu moeten we de clients dynamisch het adres van de service laten bepalen. We hadden eerder al vanuit de clients een ServiceReference gezet naar de CheckIn service (zodat o.a. de contract informatie van de service bekend is bij de client). De code die nodig is om m.b.v. WS-Discovery het adres van de service te bepalen is te zien in het volgende codevoorbeeld.

| Onderdeel | Omschrijving |
|---|---|
| UdpDiscoveryEndpoint | Dit is het type endpoint waarmee een service en een client uitgerust worden om er voor te zorgen dat de client WS-Discovery requests kan versturen en de service deze kan ontvangen. |
| UdpAnnouncementEndpoint | Dit is het type endpoint waarmee een service uitgerust moet worden om om er voor te zorgen dat de service meldt dat hij online komt of offline gaat. |
| ServiceDiscoveryBehavior | Dit behavior biedt de implementatie van het WS-Discovery protocol. Een service zal geconfigureerd worden met dit behavior om het 'discoverable' te maken. |
| DiscoveryClient | Client die gebruikt kan worden om WS-Discovery requests te versturen. |
| AnnouncementService | Een service die gebruikt kan worden om meldingen van het online komen en offline gaan van services te ontvangen. |
| DiscoveryProxy | Abstract baseclass voor de implementatie van een proxy voor serviceannouncements. |

TABEL 2.

```
using System;
using System.ServiceModel.Discovery;
using System.ServiceModel;

public partial class DiscoveryTest
{
 public void StartAnnouncementListener()
 {
  AnnouncementService announcementService = new Announcement-
  Service();
  announcementService.OnlineAnnouncementReceived += OnOnlineEvent;
  announcementService.OfflineAnnouncementReceived += OnOfflineEvent;
  using (ServiceHost announcementServiceHost =
  new ServiceHost(announcementService))
  {
   announcementServiceHost.AddServiceEndpoint(new UdpAnnouncement-
   Endpoint());
   announcementServiceHost.Open();
   Console.WriteLine("Listening for service announcements...");
   Console.WriteLine("Press any key to quit.");
   Console.ReadKey(true);
  }
 }

 private void OnOnlineEvent(object sender, AnnouncementEventArgs e)
 {
  Console.WriteLine();
  Console.WriteLine("Received an online announcement from:\n  {0}",
      e.EndpointDiscoveryMetadata.Address);
 }

 private void OnOfflineEvent(object sender, AnnouncementEventArgs e)
 {
  Console.WriteLine();
  Console.WriteLine("Received an offline announcement from \n  {0}",
      e.EndpointDiscoveryMetadata.Address);
 }
}
```

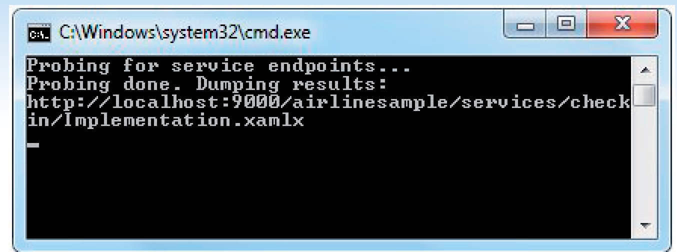
We zien dat er eerst een 'DiscoveryEndpoint' wordt gemaakt. Als adres wordt 'IPv4Multicast' gebruikt hetgeen een UDP broadcast betekent. Vervolgens wordt met dit endpoint een 'Discovery-Client' gemaakt. Daarna worden de criteria op basis waarvan we naar een service willen zoeken gespecificeerd. We zoeken in dit geval op basis van het contract van de CheckIn service. Vervolgens roepen we de 'Find' methode aan op de DiscoveryClient en dit levert een 'FindResponse' op. Voor demonstratiedoeleinden drukken we van alle gevonden endpoints het endpointaddress even af. Het resultaat is te zien in Figuur 6.

De client kan na het proben een adres uit het FindResponse gebruiken om de service aan te roepen.

## Dynamic Endpoint

In plaats van dat we de DiscoveryClient in code gebruiken is er ook een Dynamic Endpoint beschikbaar. Als dit endpoint wordt gebruikt door een client zal bij de eerste aanroep naar de service automatisch een WS-Discovery probe worden uitgevoerd. Het eerste adres dat wordt geretourneerd wordt vervolgens gebruikt om de service aan te roepen. Dit is vooral nuttig in situaties waarbij een service niet vanuit code wordt aangeroepen (bijvoorbeeld in een workflow). In dit codevoorbeeld is een configuratie te zien waarbij een Dynamic Endpoint wordt gebruikt.

```
<configuration>
  <system.serviceModel>
    <client>
      <endpoint binding="basicHttpBinding"
        contract="CheckInServiceRef.ICheckInService"
        name="CheckInServiceDynamic" kind="dynamicEndpoint" />
    </client>
  </system.serviceModel>
</configuration>
```




FIGUUR 6: DISCOVERY CLIENT RESULTAAT

In het voorgaande voorbeeld hebben we gezien dat de client op het moment dat de service aangesproken moet worden een UDP broadcast stuurt en de service daarop antwoord. Wat het WS-Discovery protocol tevens ondersteund is dat een service zich aanmeldt als hij online komt en afmeldt als hij weer offline gaat. Aan de servicekant moeten we hiervoor het DiscoveryBehavior uitbreiden. Dit hebben we gedaan voor de CheckIn service. De configuratie is te zien in dit codevoorbeeld.

```
<serviceBehaviors>
  <behavior>
    <serviceDiscovery>
      <announcementEndpoints>
        <endpoint name="UDPAnnouncement" kind="udpAnnouncement
        Endpoint" />
      </announcementEndpoints>
    </serviceDiscovery>
  </behavior>
</serviceBehaviors>
```

We zien dat er een endpoint van het type 'udpAnnouncementEndpoint' wordt toegevoegd aan het 'DiscoveryBehavior'. Dit is alles wat me moeten doen om er voor te zorgen dat de service zich aanmeldt en afmeldt via een UDP broadcast. Met een dergelijke broadcast wordt de DiscoveryMetadata van de service verstuurd waarin, zoals we eerder hebben kunnen zien, ook het adres van de service is opgenomen. Wanneer een client deze meldingen zou opvangen en opslaan in een cache, kan deze op het moment dat een service aangeroepen moet worden het adres van de service uit de cache lezen en de service aanroepen. Voor het ontvangen van announcements biedt WCF de 'AnnouncementService' class. Deze klasse gebruiken we in de voorbeeldcode:

```
Console.WriteLine("Probing for service endpoints...");
DiscoveryEndpoint discoEndpoint =
  new UdpDiscoveryEndpoint(UdpDiscoveryEndpoint.DefaultIPv4Multi-
  castAddress);
DiscoveryClient discoClient = new DiscoveryClient(discoEndpoint);
FindCriteria criteria = new FindCriteria(typeof(ICheckInService));
FindResponse findResponse = discoClient.Find(criteria);
Console.WriteLine("Probing done. Dumping results:");
foreach (EndpointDiscoveryMetadata metadata in findResponse.End-
points)
{
 Console.WriteLine(metadata.Address.ToString());
}
```

Hierin zien we dat een instantie wordt gemaakt van de AnnouncementService en dat daaraan eventhandlers worden gekoppeld voor de Online- en OfflineAnnouncementReceived events. Vervolgens wordt de service gehost via een UdpAnnouncementEndpoint en gaat de service wachten op announcements. In de eventhandlers wordt een melding afgedrukt wanneer er een online of offline announcement wordt ontvangen. Hierbij wordt het endpointaddress uit de DiscoveryMetadata afgedrukt. Als we een client starten en de code uitvoeren gaat deze netjes staan wachten op announcements.

Als we vervolgens de CheckIn service activeren (door bijvoorbeeld een keer de WSDL op te halen), zien we een online melding binnenkomen. Dit resultaat is te zien in Figuur 7.

Een belangrijk punt om hieruit mee te nemen is dat een service dus eerst een keer moet worden aangeroepen alvorens hij een online announcement stuurt. Eén van de oplossingen die hiervoor in de praktijk wordt ingezet, is dat de service wordt uitgerust met een soort Ping operatie en dat er een achtergrondproces (bijvoorbeeld een Windows service) draait van waaruit op vaste intervallen de Ping operatie op de service aanroept. Dit is iets om rekening mee te houden als gekozen wordt om met announcements te werken.

## DiscoveryProxy

Het anouncement mechanisme werkt op zich prima. Het is echter niet wenselijk dat alle clients zelf een announcement service moeten hosten om hiervan gebruik te maken. Nu is het zo dat Discovery binnen WCF 4 twee modi kent: AdHoc en Managed. Wat we tot nu toe gezien hebben is AdHoc mode (clients en services broadcasten respectievelijk hun probes en announcements). In Managed mode maken we gebruik van een DiscoveryProxy. Een voordeel van Managed mode t.o.v. AdHoc mode is dat Managed mode over meerdere subnetten heen functioneert omdat het niet met UDP broadcasts werkt. Services sturen hun announcements rechtstreeks naar de proxy en de clients vragen rechtstreeks aan de DiscoveryProxy om de locatie van een service, beide via bijvoorbeeld een BasicHttpBinding. Om deze functionaliteit te kunnen realiseren worden er in WCF 4 verschillende classes geleverd (waaronder bijvoorbeeld een DiscoveryProxy base class). We zullen dit mechanisme niet verder uitwerken in dit artikel. De Microsoft samples die beschikbaar zijn voor .NET Framework 4 Bèta 2 (zie de lijst met links) bevatten een voorbeeld van dit mechanisme.

## Routing services

Een andere manier om clients onafhankelijk van de locatie van de services te maken is door gebruik te maken van de Routing Services feature in WCF 4. Routing services stelt ons in staat een Routing service te maken en die te configureren met regels op basis waarvan een request dat van een client komt gerouteerd wordt naar een bepaalde service. De regels (filters genoemd) worden in de configuratie van de Routing service gespecificeerd. Dit kan – zoals gebruikelijk bij WCF - zowel in code als in een config file. De configuratie bestaat grofweg uit drie onderdelen:

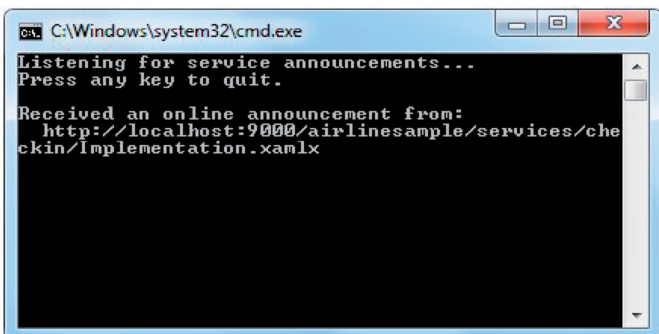
- Een set met client endpoints waarin een endpoint is opgenomen voor elke service waar verkeer naartoe gerouteerd moet kunnen worden.
- Een set met filters. Elk filter bevat een criterium dat tegen een request aangehouden kan worden om te zien of het request hieraan voldoet.
- Een filtertabel waarin een koppeling wordt gelegd tussen filters en client endpoints.

Wanneer een request de Routing service bereikt, zal elk filter tegen het request worden aangehouden om te kijken of het request voldoet aan het criterium van het filter. Een voorbeeld van een criterium dat kan worden gebruikt om een filter te definiëren is de SOAP Action van het request (we zien later welke typen criteria er nog meer zijn). Als een match is gevonden, zal het request worden doorgestuurd naar het client endpoint dat in de filtertabel is gekoppeld aan het desbetreffende filter. In Figuur 8 is dit schematisch weergegeven.

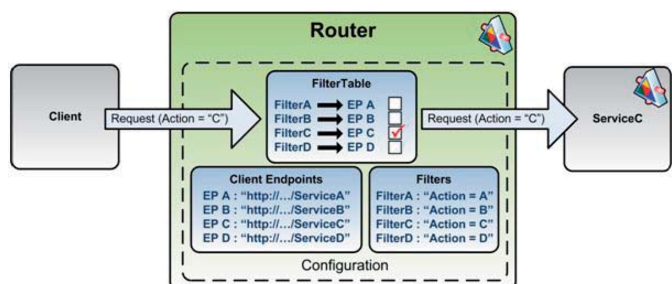
Een eventueel response van de achterliggende service wordt door de Routing service geretourneerd aan de client. Het kan natuurlijk voorkomen dat een request geen enkele match met een filter oplevert. In dat geval zal de Routing service een fout retourneren aan de client. Het is ook mogelijk dat een request meer dan één match oplevert. In dat geval zal het request worden doorgestuurd naar meerdere achterliggende services. Dit is echter alleen mogelijk voor one-way en duplex verkeer.

We zullen de Routing Services feature demonstreren aan de hand van onze voorbeeldoplossing, waarbij de Routing service door de clients worden gebruikt om de CheckIn service aan te roepen. Allereerst introduceren we een nieuwe service: de Routing service. Dit is een standaard WCF service waarbij de implementatie van de service wordt geleverd door de .NET Framework class System.ServiceModel.Routing.RoutingService. Het configureren van de Routing service doen we in de web.config die is weergegeven in dit codevoorbeeld.

```
<configuration>
  <system.web>
    <compilation
      debug="true"
      targetFramework="4.0">
      <assemblies>
        <add assembly="System.ServiceModel.Routing, Version=4.0.0.0,
          Culture=neutral, PublicKeyToken=31bf3856ad364e35"/>
      </assemblies>
    </compilation>
  </system.web>
  <system.serviceModel>
    <services>
      <service
        behaviorConfiguration="RoutingServiceBehaviors"
        name="System.ServiceModel.Routing.RoutingService">
        <endpoint
          address="http://localhost:9999/airlinesample/services/
```




FIGUUR 7:ANNOUNCEMENTSERVICE RESULTAAT




FIGUUR 8: SCHEMATISCH OVERZICHT ROUTINGSERVICE

```
        routing/service.svc"
        binding="basicHttpBinding"
        name="reqReplyEndpoint"
        contract="System.ServiceModel.Routing.IRequestReplyRouter" />
    </service>
  </services>
  <behaviors>
    <serviceBehaviors>
      <behavior name="RoutingServiceBehaviors">
        <serviceDebug includeExceptionDetailInFaults="true" />
        <serviceMetadata httpGetEnabled="True"/>
        <routing filterTableName="AirlineSampleRoutingTable" />
      </behavior>
    </serviceBehaviors>
  </behaviors>
  <client>
    <endpoint
      name="CheckInService"
      address="http://localhost:9000/airlinesample/services/
      checkin/implementation.xamlx"
      binding="basicHttpBinding"
      contract="*" />
  </client>
  <routing>
    <filters>
      <filter name="StartCheckin" filterType="Action"
        filterData="http://net40.airlinesample/services/checkin/
        startcheckin" />
      <filter name="ChangeSeat" filterType="Action"
        filterData="http://net40.airlinesample/services/checkin/
        changeseat" />
      <filter name="ConfirmSeat" filterType="Action"
        filterData="http://net40.airlinesample/services/checkin/
        confirmseat" />
      <filter name="PrintBoardingPass" filterType="Action"
        filterData="http://net40.airlinesample/services/checkin/
        printboardingpass" />
    </filters>
    <filterTables>
      <filterTable name="AirlineSampleRoutingTable">
        <add filterName="StartCheckin" endpointName="CheckIn-
        Service" />
        <add filterName="ChangeSeat" endpointName="CheckInService" />
        <add filterName="ConfirmSeat" endpointName="CheckIn-
        Service" />
        <add filterName="PrintBoardingPass" endpointName="CheckIn-
        Service" />
      </filterTable>
    </filterTables>
  </routing>
 </system.serviceModel>
</configuration>
```

We zien de definitie van de service met als implementatie de RoutingService. Daarnaast zien we dat er een speciaal endpoint wordt toegevoegd aan de service. Het contract van het endpoint bepaalt hier welke typen request de Routing service kan verwerken. In dit geval gaat het om standaard request-reply verkeer. Naast IRequestReplyRouter zijn ook ISimplexDatagramRouter, ISimplexSessionRouter, en IDuplexSessionRouter beschikbaar voor respectievelijk one-way, session en duplex communicatie. We zien verder dat de service is uitgerust met een aantal servicebehaviors. Onder deze behaviors zien we een behavior dat nieuw is in WCF 4 genaamd 'routing'. Dit behavior zorgt ervoor dat de Routing service zijn werk kan doen. Hier wordt gespecificeerd dat de filtertabel genaamd 'AirlineSampleFilterTable' moet worden gebruikt. Zoals eerder beschreven bevat de filtertabel koppelingen tussen filters en client endpoints. De client endpoints definiëren we zoals we gewend zijn bij WCF in de 'client' sectie van de web.config. Bij de client endpoints vullen we als contract altijd '*' in. De Routing service zorgt ervoor dat uiteindelijk het juiste contract gebruikt wordt. De filters en de filtertabel definiëren we in een speciale sectie genaamd 'routing'. In onze voorbeeldoplossing zullen we routeren op basis van de SOAP Action in het bericht, dus we gebruiken hier alleen het filtertype 'Action'. We zien dat er een filter is opgenomen voor de SOAP Action van elke operatie die door de CheckIn service wordt aangeboden. In de filtertabel is vervolgens aan elk filter de bijbehorende service gekoppeld. Onze Routing service is nu gereed om ingezet te worden. We veranderen vervolgens het adres dat de clients gebruiken om de CheckIn service aan te roepen in het adres van de Routing service. Wanneer we nu een client starten zien we geen verschil in de werking ervan. Hoe kunnen we nu controleren of de Routing service daadwerkelijk wordt gebruikt? We kunnen hiervoor WCF tracing gebruiken. Wanneer tracing wordt geactiveerd voor de Routing service, zal deze extra tracing informatie opleveren om te achterhalen wat de Routing service aan het doen is. Deze tracing kan bekeken worden m.b.v. de standaard WCF Trace Viewer.

## Filtertypen

Naast het filtertype 'Action' dat we in het voorgaande voorbeeld hebben gezien is er nog een aantal filtertypen beschikbaar. Dit zijn o.a.:

- MatchAll - Elk request zal een match opleveren.
- And - Filter op 2 filters (die via de logische AND operator worden gecombineerd).
- XPath - Filter op basis van het resultaat van een XPath query die wordt losgelaten op het complete request.
- Custom - Mogelijkheid om een eigen filter te maken.

Vooral de XPath filters zijn erg krachtig. Deze maken het mogelijk om te routeren op basis van gegevens in het request. Via een XPath expressie kan een XML node in het request worden aangewezen. Een Action filter kan bijvoorbeeld worden nagemaakt door een filter van het type XPath te gebruiken en vervolgens de 'Action' header op te zoeken met een XPath query. In codevoorbeeld 8 zien we hiervan een voorbeeld. De namespace prefixen die in de XPath expressies worden gebruikt zijn gedefinieerd in een namespacetabel. Wanneer we eigen headers toevoegen aan een request, kunnen ook deze gebruikt worden om te routeren. Het is een kwestie van de namespace waarin de headers zich bevinden opnemen in de namespacetable en vervolgens de juiste XPath query gebruiken.

```
<routing>
  <namespaceTable>
    <add prefix="s" namespace="http://schemas.xmlsoap.org/soap/
    envelope/" />
    <add prefix="wsa" namespace="http://schemas.microsoft.com/
    ws/2005/05/addressing/none"/>
    <add prefix="msg" namespace="http://net40.airlinesample/schema" />
    <add prefix="hdr" namespace="http://net40.airlinesample/headers" />
  </namespaceTable>
  <filters>
    <filter name="ActionFilterSimluator" filterType="XPath"
      filterData="/s:Envelope/s:Header/wsa:Action =
          'http://net40.airlinesample/services/checkin/startcheckin'" />

    <filter name="SilverCustomerHeaderFilter" filterType="XPath"
       filterData="/s:Envelope/s:Header/hdr:CustomerType = 'Silver'"/>
    <filter name="GoldCustomerHeaderFilter" filterType="XPath"
       filterData="/s:Envelope/s:Header/hdr:CustomerType = 'Gold'"/>

    <filter name="FrequentFlyerFilter" filterType="XPath"
       filterData="//msg:ProgramMembership = 'FrequentFlyer'"/>
  </filters>
</routing>
```

In het voorbeeld wordt ervan uitgegaan dat er een header genaamd 'CustomerType' aan het request is toegevoegd dat als waarde 'Silver' of 'Gold' kan hebben. Het is ook mogelijk om op basis

van gegevens in de body van het request te routeren. Hiervoor moet wel iets speciaals gebeuren. Standaard is het namelijk alleen mogelijk om in de headers van een request te kijken en niet in de body (dit is vanwege performance redenen). Om ook de inhoud van de body beschikbaar te maken voor de Routing service, moet het routing behavior hiervoor worden geconfigureerd. Dit behavior heeft een property genaamd 'routeOnHeadersOnly'. Als we deze property de waarde 'false' geven, kunnen we ook gegevens uit de body gebruiken voor het routeren (zie het 'FrequentFlyerFilter' in codevoorbeeld 8). In het voorbeeld wordt ervan uitgegaan dat de body van het request een veld bevat genaamd 'ProgramMembership' dat als waarde 'None' of 'FrequentFlyer' kan hebben. Een nadeel bij routeren op gegevens in de body is dat elk bericht nu moet worden gebufferd door de Routing service om in de body te kunnen kijken (in plaats van het bericht als stream te kunnen behandelen). Dit heeft uiteraard impact op de performance.

## Alternatieve endpoints

Het kan natuurlijk voorkomen dat er een fout optreedt wanneer een request wordt doorgestuurd naar de achterliggende service. Het is echter mogelijk om alternatieve client endpoints te specificeren in de Routing service configuratie. We specificeren dan meerdere client endpoints in de configuratie en definieren een 'backuplist' waarin we verwijzen naar 1 of meerdere alternatieve client endpoints. Bij elk filter in een filtertable kan vervolgens een backuplist worden gespecificeerd. In het geval dat er een exception optreedt bij het aanroepen van de achterliggende service, zal het request worden doorgestuurd naar het alternatieve endpoint. Wanneer in de lijst met alternatieve endpoints meerdere endpoints worden gespecificeerd, zal de Routing service deze allemaal aflopen totdat

- een alternatieve achterliggende service het request succesvol heeft verwerkt of
- alle endpoints in de lijst een keer zijn geprobeerd.

In het laatste geval zal uiteindelijk een foutmelding worden geretourneerd aan de client.

Protocol bridging

Een Routing service is in staat om een ander protocol te gebruiken voor het aanroepen van de achterliggende service dan is gebruikt voor het aanroepen van de Routing service. Een client kan bijvoorbeeld de Routing service aanroepen via een BasicHTTPBinding. De Routing service kan vervolgens de achterliggende service aanroepen via een NetTCP binding (mits de achterliggende service deze binding ondersteunt uiteraard).

## Dynamische updates

Het configureren van de Routing service op de hierboven beschreven manier werkt prima. Toch zien we in de praktijk vaak dat er dingen veranderen in het servicelandschap. Services worden om wat voor reden dan ook verplaatst naar andere servers, er worden instanties van een service toegevoegd, etc. Het is natuurlijk belangrijk dat een Routing service deze veranderingen aankan. Nu kan natuurlijk de configuratie worden aangepast en de Routing service herstart worden. Dit kan echter gevolgen hebben voor lopende zaken en zorgt ervoor dat gedurende een korte periode de Routing service niet beschikbaar is. Aangezien clients zonder de Routing service geen services aan kunnen roepen, kan dit tot verstoringen leiden. De routingservices in WCF 4 bieden hier echter een oplossing voor. Als de Routing service wordt gestart wordt er automatisch een serviceextension van het type RoutingExtension geregistreerd bij de servicehost. Deze extension heeft een methode ApplyConfiguration die als argument een RoutingConfiguration instantie verwacht. Als deze methode wordt aangeroepen op de routingextension van een draaiende Routing service, zal de nieuwe configuratie van kracht worden. Requests die op dat moment nog actief zijn zullen echter gewoon afgehandeld worden volgens de originele configuratie. Een manier om van dit mechanisme gebruik te maken is zelf een serviceextension te maken die reageert op een bepaald extern event en vervolgens de ApplyConfiguration methode aanroept op de RoutingExtension en daarbij de nieuwe configuratie meegeeft. Enkele voorbeelden van externe events die hiervoor gebruikt kunnen worden zijn: een WCF 4 Discovery announcement, een SQL Event notification, een FileSystemWatcher event, een Cache event, enz. In codevoorbeeld 9 is een voorbeeld te zien van hoe een Routing configuratie in code kan worden opgebouwd.

```
ContractDescription contract =
  ContractDescription.GetContract(typeof(IRequestReplyRouter));
BasicHttpBinding binding = new BasicHttpBinding();
RoutingConfiguration routerConfig = new RoutingConfiguration();
string address = "http://localhost/services/someservice.svc";
ServiceEndpoint endpoint =
  new ServiceEndpoint(contract, binding, new EndpointAddress
  (address));
IList<ServiceEndpoint> endpoints = new List<ServiceEndpoint>() {
endpoint };
string[] actions = new string[1];
actions[0] = "http://localhost/services/someservice/someaction"
List<string> actions = new List<string>();
ActionMessageFilter filter = new ActionMessageFilter(actions);
routerConfig.FilterTable.Add(filter, endpoints);
```

## Tot slot

Wij zijn van mening dat de nieuwe mogelijkheden die WCF en WF ons gaan bieden in versie 4 ons in staat stellen om verschillende uitdagingen die we vaak tegenkomen in het veld op een elegante manier op te kunnen lossen. In dit artikel hebben we een overzicht gegeven van verschillende nieuwe features. We raden iedereen aan om met de samples die door Microsoft worden geleverd voor het .NET Framework 4 Bèta 2 (zie de lijst met links) aan de slag te gaan.

## Links:

WS-Discovery specificatie:
http://specs.xmlsoap.org/ws/2005/04/discovery/ws-discovery.pdf
Visual Studio 2010 & .NET 4 website:
http://go.microsoft.com/link/?LinkID=151797
The .Net Endpoint (WCF / WF team blog):
http://blogs.msdn.com/endpoint/default.aspx
WCF Discovery team blog:
http://blogs.msdn.com/discovery/

**Willem Meints,** is IT Consultant bij Info Support. Hij is bereikbaar op: willemm@infosupport.com.

**Alexander Molenkamp,** is IT Consultant bij Info Support. Hij is bereikbaar op: sanderm@infosupport.com.

**Edwin van Wijk,** is IT Architect bij Info Support. Hij is bereikbaar op: edwinw@infosupport.com.