Gijs de Jong is principal consultant bij Microsoft Services

WCF servicemodel internals & extensibility

EIGEN EXTENSIES EN BEHAVIORS BOUWEN

Windows Communication Foundation (WCF), beter bekend onder de codenaam 'Indigo', is in voorgaande edities van het .NET Magazine al uitvoerig aan bod gekomen. In een aantal artikelen is uiteengezet hoe je met WCF veilige en robuuste servicegeoriënteerde oplossingen kunt creëren. "De rijke infrastructuur die WCF biedt, is zeer configurabel en zal in de meeste gevallen voorzien in de behoefte van de ontwikkelaar." In dit artikel gaat de auteur in op de uitbreidbaarheid van het WCF-programmeermodel als de standaardmogelijkheden niet toerijkend zijn.

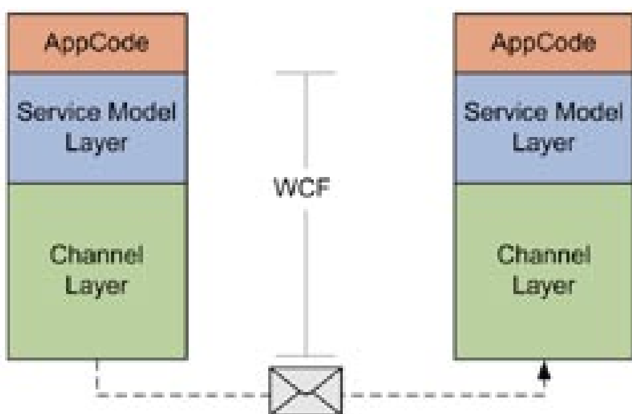
WCF biedt een rijk platform voor het bouwen van gedistribueerde oplossingen. Er zijn standaardfaciliteiten beschikbaar voor bijvoorbeeld beveiliging, betrouwbare communicatie en transactionele verwerking. Ook biedt WCF een heel scala aan mogelijkheden om het runtime gedrag van een client en service te beïnvloeden. Al deze mogelijkheden kunnen dusdanig geconfigureerd worden, zodat ze aansluiten op de eisen van de te realiseren oplossing. Het kan voorkomen dat een zeer specifieke oplossing bepaalde vereisten heeft die niet standaard geboden worden door WCF. In zulke situaties moet het mogelijk zijn de standaardfunctionaliteit van WCF op relatief eenvoudige wijze aan te vullen of te vervangen door een eigen implementatie. Voordat we inhoudelijk ingaan op alle uitbreidingsmogelijkheden van WCF, staan we eerst kort stil bij een aantal architectuurprincipes.

WCF Architectuur

De WCF-architectuur kan grofweg in twee lagen opgesplitst worden, namelijk de channel-laag en de servicemodellaag; zie afbeelding 1. De channel-laag zorgt voor de communicatie-infrastructuur en het servicemodel voor een objectgeoriënteerd en declaratief programmeermodel.

Channel Layer

De channel-laag is de onderliggende communicatie-infrastructuur van WCF. Deze laag bestaat minimaal uit een transportchannel en een encoder. Het transportchannel implementeert het daadwerkelijke communicatieprotocol, zoals http, tcp, named pipes of MSMQ. De encoder zorgt voor de codering en decodering van de berichten op het moment dat ze verstuurd en ontvangen worden. WCF ondersteunt standaard drie type encoders; text/xml, binary en MTOM.



Afbeelding 1. WCF-architectuurlagen

Optioneel voorziet de channel-laag in channels die (WS*)-protocollen toevoegen aan de communicatie-infrastructuur, zoals WS-Security, WS-ReliableMessaging, enzovoort. De channel-laag wordt geconfigureerd door bindings. Een binding is een samenstelling van een transportchannel, een encoder en nul of meer protocolchannels. WCF biedt standaard een aantal voorgedefinieerde bindings. Deze systeem-bindings bieden een eenvoudige manier van configuratie voor de meest voorkomende scenario's. De channel-laag is volledig configurabel en uitbreidbaar. Het is mogelijk om met eigen gedefinieerde bindings nieuwe scenario's te ondersteunen op basis van bestaande transportchannels, encoders en protocolchannels. Op deze manier kunnen bedrijfsspecifieke standaarden afgedwongen worden. Op security-gebied zou bijvoorbeeld afgedwongen kunnen worden dat voor intranetscenario's altijd Kerberos als authenticatieprotocol wordt gebruikt en voor internetscenario's X509-certificaten. Mochten de standaard beschikbare onderdelen niet toereikend zijn voor een bepaald scenario, dan is het mogelijk eigen transportchannels, encoders en protocolchannels te bouwen. Dit zou een uitkomst kunnen bieden wanneer er met een legacy-systeem geïntegreerd moet worden dat geen ondersteuning biedt voor webservices.

Service Model Layer

De servicemodellaag biedt het programmeermodel voor de WCF-ontwikkelaar. Deze laag is verantwoordelijk voor de vertaalslag van het objectgeoriënteerde en declaratieve programmeermodel naar de wereld van berichten in de channel-laag. In de servicemodellaag kan het interne runtime-gedrag van zowel de client als de service beïnvloed worden door zogenaamde 'behaviors'. WCF biedt standaard een uitgebreide verzameling behaviors. Hiermee kan bijvoorbeeld de instantiatie van services, concurrency (parallelle benadering) en autorisatie worden geconfigureerd. Afhankelijk van het behavior kunnen deze met behulp van code en/of configuratie toegepast worden. Het servicemodel is uitbreidbaar door bestaande behaviors te

vervangen of aan te vullen met eigen geprogrammeerde behaviors. Behaviors stellen de programmeur in staat op verschillende punten in het servicemodel interceptie toe te passen. Deze faciliteit kan vervolgens gebruikt worden om bijvoorbeeld inspectie, logging en transformatie van berichten en/of parameters uit te voeren. In de rest van dit artikel gaan we dieper in op het zelf bouwen van behaviors.

WCF-initialisatieproces

Voor het bouwen van een op WCF-gebaseerde applicatie is er altijd een aantal basisconcepten waar elke ontwikkelaar mee te maken krijgt. Zo bevat een service een of meer endpoints. Een endpoint is een locatie op het netwerk waar berichten naar toe gestuurd kunnen worden. Elk endpoint is opgebouwd uit een *address, binding* en *contract*, de zogenaamde 'ABC' van WCF; zie afbeelding 2.

- **Address –** waar moet een bericht naar toe gestuurd worden?
- **Binding –** hoe moet een bericht verstuurd worden; transportprotocol, encoding, WS*-protocollen?
- **Contract –** wat kan een service; beschrijving van de operaties, data- en berichtformaten?

Deze drie elementen worden in een combinatie van code (C#, Visual Basic, et cetera) en configuratie beschreven. De contracten en service-implementatie worden altijd in code uitgedrukt, het adres en de binding normaliter in configuratie; zie codevoorbeeld 1.

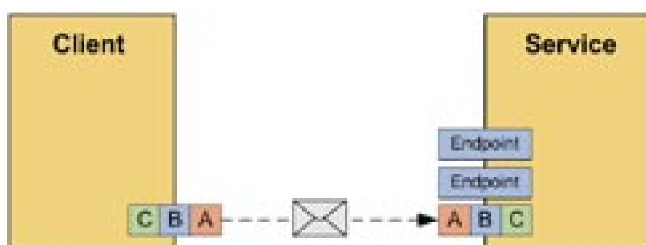
Een WCF-applicatie bestaat dus uit een combinatie van code en configuratie, maar welke stappen worden er precies uitgevoerd om tot een draaiende applicatie te komen?

1. **Codereflectie en laden van configuratie –** dit proces vindt plaats zodra er een instantie van het ChannelFactory-object (client-side) of het ServiceHost-object (service-side) wordt gecreëerd.
2. **Description –** op basis van de vorige stap wordt een zogenaamde description opgebouwd. Een description is een object dat de WCF-applicatie beschrijft.
3. **Runtime initialisatie –** zodra de *Open*-methode op de ServiceHost (service-side) of ChannelFactory (client-side) aangeroepen wordt, zal op basis van het description-object de runtime opgebouwd worden en worden de desbetreffende resources geopend.

In de hieronder volgende secties gaan we dieper in op deze drie stappen.

Configuratie en codereflectieproces

Het laden van configuratie en het reflecteren over de code gebeurt zodra er een nieuwe instantie van het object ServiceHost of het object ChannelFactory wordt gecreëerd. Het ServiceHost-type is verantwoordelijk voor het hosten van een service. Dit type zullen we expliciet moeten instantiëren wanneer we met een *self-hosted* scenario (buiten IIS/Windows Activation Service) te maken hebben. Het ChannelFactory-object is op het eerste gezicht iets minder zichtbaar, maar wordt onder water gebruikt door de gegenereerde proxy op de client. Dit object wordt geïnstantieerd zodra er een nieuwe instantie gecreëerd wordt van het proxy-object.



Afbeelding 2. 'ABC' van WCF

De codeanalyse en het laden van de configuratie kan uitgebreid of vervangen worden door een eigen implementatie. De reflectie over de service en bijbehorende contracten (Service-, Message- en DataContract-attributen) zal over het algemeen minder snel aangepast worden. Het beïnvloeden van hoe de configuratie wordt geladen, is eerder een kandidaat. Een toepassing zou kunnen zijn om de configuratie niet uit de standaard app.config te laden, maar bijvoorbeeld uit een centrale configuratiedatabase.

Als we het configuratieproces van een service willen beïnvloeden, zullen we een eigen servicehost moeten maken. Dit is eenvoudig te realiseren door te erven van het *ServiceHost*-type en de methode *ApplyConfiguration* te overschrijven. In *ApplyConfiguration* kan de configuratie van een willekeurige locatie ingelezen worden. Op basis van deze configuratie zal het description-object (*this.Description*) zelf handmatig aangevuld moeten worden met endpoints en mogelijkerwijs behaviors; zie codevoorbeeld 2.

In een *self-hosted* scenario, waarbij gebruik gemaakt wordt van bijvoorbeeld een console, winforms of een windows-service-applicatie, kan *MyCustomHost* gewoon geïnstantieerd worden; zie codevoorbeeld 3.

Wanneer gebruik gemaakt wordt van IIS of Windows Activation Service (WAS) is het uiteraard niet mogelijk zelf het *ServiceHost*-object te instantiëren, aangezien IIS dit automatisch voor ons doet. Om dan toch gebruik te kunnen maken van het eigen

```
// Service Contract
[ServiceContract]
public interface IBankingService
{
   [OperationContract]
   string TransferMoney(string creditAccount, string  debitAccount,
double amount);
}

// Service Implementatie
public class BankingService : IBankingService
{
   public string TransferMoney(string creditAccount, string
debitAccount, double amount)
   {
      string result = string.Format("{0} dollars has been
      transferred", amount, debitAccount, creditAccount);

      Console.WriteLine(result);
      return result;
   }
}

// Service configuratie (address & binding)
<configuration>
   <system.serviceModel>
      <services>
         <service name="BankingService">
            <endpoint
               address="http://localhost:9000"
               binding="wsHttpBinding"
               contract="IBankingService"/>
         </service>
      </services>
   </system.serviceModel>
</configuration>
```

Codevoorbeeld 1. WCF-service-implementatie en configuratie

gedefinieerde host-type zal een bijbehorende *ServiceHostFactory* gebruikt moeten worden. Deze hoeft slechts een instantie van de custom servicehost te retourneren; zie codevoorbeeld 4.
De custom ServiceHostFactory kan vervolgens geconfigureerd worden in het svc- bestand van IIS/WAS; zie codevoorbeeld 5.
Door een eigen ServiceHost en optioneel een eigen Service-HostFactory te gebruiken kan dus invloed uitgeoefend worden op het laden van de configuratie. Je kunt nog een stap verder gaan en het codereflectieproces vervangen. Om dit te realiseren zal ook weer een eigen ServiceHost en ServiceHostFactory gemaakt moeten worden, maar die zullen nu moeten erven van *ServiceHostBase* en *ServiceHostFactoryBase*. Hoe dit gerealiseerd kan worden, ligt buiten de scope van dit artikel.

Description

Het resultaat van het inlezen van de configuratie en het code-reflectieproces is een zogenaamd description-object. Het description-object bevat een beschrijving van alle endpoints en behaviors van een client of service; zie afbeelding 3. Deze beschrijving dient in een later stadium als basis om de runtime op te bouwen.
Na het creëren van een instantie van de ServiceHost en/of ChannelFactory is het mogelijk het description-object uit te lezen. In codevoorbeeld 6 worden alle endpoints van een service in de console getoond.
Indien gewenst kan op dit punt het description-object nog gewoon aangepast worden. Op het moment dat expliciet de methode *Open* op de ServiceHost wordt aangeroepen, zal de runtime ook daadwerkelijk opgebouwd worden en kunnen er geen wijzigingen meer plaatsvinden op het description-object.

Aan de kant van de client wordt het openen van de ChannelFactory impliciet gedaan wanneer voor de eerste keer een operatie op de proxy wordt aangeroepen. Het is ook mogelijk dit expliciet te doen door *Open* aan te roepen op de property ChannelFactory van het proxy-object.

Runtime initialisatie

Bij het instantiëren van de ServiceHost en/of ChannelFactory wordt de code geanalyseerd en de configuratie ingeladen, met als resultaat een description-object. Zodra *Open* aangeroepen wordt op de ServiceHost en/of ChannelFactory, zal op basis van het description-object de runtime opgebouwd worden en de desbetreffende resources geopend worden. Vervolgens zullen de desbetreffende resources, zoals een TCP/IP-poort, geopend worden. De runtime-architectuur wordt in afbeelding 4 weergegeven. Op servicemodelniveau zijn er twee objecten die verantwoordelijk zijn voor de interne werking; de *proxy* aan de clientzijde, en de *dispatcher* aan de servicezijde.
Zowel de proxy als de dispatcher bevatten tal van uitbreidings-mogelijkheden om op verschillende punten eigen functionaliteit toe te voegen. Deze uitbreidingen worden extensies genoemd.
In de volgende paragraaf zullen we in meer detail kijken naar de verschillende typen extensies en hoe deze aan de proxy en dispatcher toegevoegd kunnen worden.

Uitbreiden van de proxy (client)

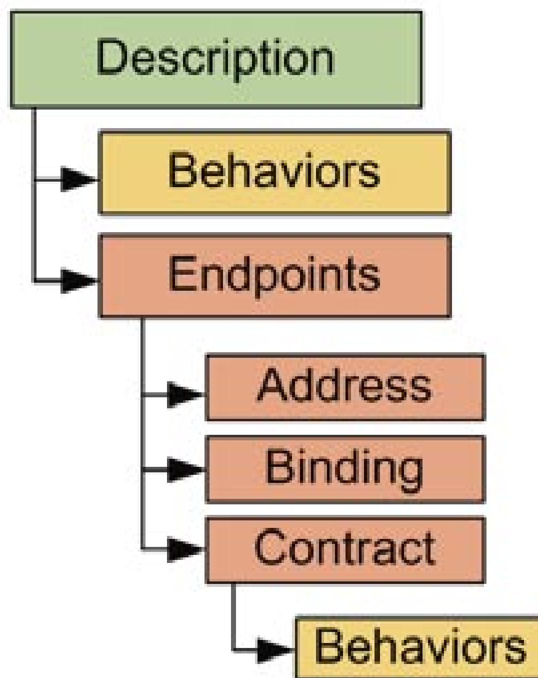
De proxy is verantwoordelijk voor de conversie van een methodeaanroep vanuit de clientapplicatiecode naar een bericht, dat vervolgens door de channel-laag verstuurd wordt naar de

```
public class MyCustomHost : ServiceHost
{
   public DerivedHost(Type t, params Uri[] baseAddresses)
                  : base(t, baseAddresses)
   { }

   protected override void ApplyConfiguration ()
   {
      // Laad de configuratie zelf en voeg endpoints (en
      // behaviors) toe aan this.Description

      // Om de app.config te laden roep dan
      // base.ApplyConfiguration(); aan.
   }
}
```
Codevoorbeeld 2. Custom ServiceHost

```
static void Main(string[] args)
{
   MyCustomHost host = new MyCustomHost(    typeof(BankingService));

   host.Open();
}
```
Codevoorbeeld 3. Custom – Gebruik van een custom ServiceHost in een self-hosted scenario

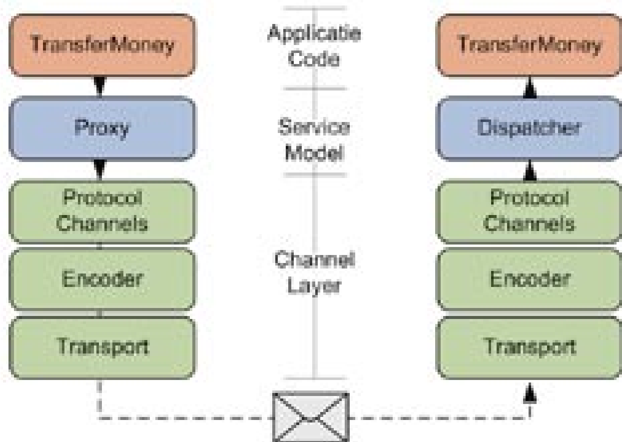
```
public class MyCustomFactory : ServiceHostFactory
{
   public override ServiceHost CreateServiceHost(
         Type t, Uri[] baseAddresses)
   {
      return new MyCustomHost(t, baseAddresses);
   }
}
```
Codevoorbeeld 4. Custom ServiceHostFactory

```
<% @ServiceHost Factory="MyCustomFactory" Service="MyService" %>
```
Codevoorbeeld 5. Custom ServiceHostFactory in IIS/WAS-scenario (svc-bestand)



Afbeelding 3. Description-object

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

foreach (ServiceEndpoint endpoint in host.Description.Endpoints)
{
   Console.WriteLine(endpoint.Address.Uri.ToString());
   Console.WriteLine(endpoint.Binding.Name);
   Console.WriteLine(endpoint.Contract.Name);

}
// host.Open();
```
Codevoorbeeld 6. Programmeren tegen het description-object



Afbeelding 4. Runtime-architectuur

service. Voor mogelijke responsberichten zal het proces vice versa uitgevoerd worden. In dit proces wordt een aantal stappen onderkend, zoals parameterinspectie, message-formatting, methode op operatie-mapping en message-inspectie. De concrete implementatie van een *extensiblity-hook* wordt een extensie genoemd.

Sommige *extensibility-hooks* kunnen slechts een enkele extensie bevatten, zo kan er logischerwijs maar één message-formatter zijn. Andere *extensibility-hooks* zijn geïmplementeerd als collecties, waardoor er meer extensies toegepast kunnen worden. Zo is het bijvoorbeeld mogelijk verscheidene message-inspectors toe te voegen. Afhankelijk van het type extensie kan deze betrekking hebben op slechts een individuele operatie, zoals *TransferMoney* en *GetBalance*, of op alle operaties van de proxy.

Extensies zijn eenvoudig zelf te programmeren. In eerste instantie moeten we bepalen wat voor type extensie we willen realiseren. Elk type extensie heeft een bijbehorende interface die we in een eigen gedefinieerde class moeten implementeren. In codevoorbeeld 7 realiseren we een message inspector-extensie die alle (applicatieve) berichten - die tussen client en service worden uitgewisseld - in de console toont. Voor het realiseren van een message-inspector moet de interface *System.ServiceModel.Dispatcher.IClientMessageInspector* gebruikt worden. Deze interface bevat twee methodes waarmee je het bericht voor het versturen (BeforeSendRequest) en na het ontvangen (AfterReceiveReply) kunt inspecteren of zelfs aanpassen.

Het programmeren van een custom extensie is de eerste stap. Vervolgens voegen we de extensie aan de runtime toe. Dit laatste doen we door gebruik te maken van een custom behavior. Afhankelijk van het toepassingsgebied (scope) van de behavior kan deze door code (declaratief en/of imperatief) en configuratie toegepast worden. Zo kunnen behaviors van toepassing zijn op operatie, endpoint, contract en serviceniveau.

```
class MyMessageInspector:IClientMessageInspector
{
  public object BeforeSendRequest(ref Message request,
                IClientChannel channel)
  {
    Console.WriteLine(request);
    return null; // Don't pass any correlationState
  }

  public void AfterReceiveReply(ref Message reply,
    object correlationState)
  {
      Console.WriteLine(reply);
  }
}
```

Codevoorbeeld 7. Custom message-inspector

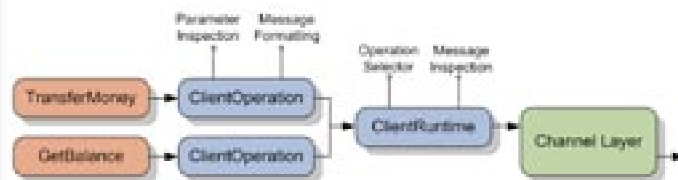
```
class MyMessageInspectorBehavior:IEndpointBehavior
{
  public void ApplyClientBehavior(ServiceEndpoint endpoint,
      ClientRuntime clientRuntime)
  {
    clientRuntime.MessageInspectors.Add(
        new MyMessageInspector());
  }

  public void AddBindingParameters(ServiceEndpoint endpoint,
      BindingParameterCollection bindingParameters)
  {}

  public void ApplyDispatchBehavior(ServiceEndpoint  endpoint,
      EndpointDispatcher endpointDispatcher)
  {}

  public void Validate(ServiceEndpoint endpoint)
  {}
}
```

Codevoorbeeld 8. Custom endpoint-behavior

Voor het voorbeeld van de message-inspector gebruiken we een zogenaamd endpoint-behavior. Dit betekent dat we alle (applicatieve) berichten die via het desbetreffende endpoint worden verstuurd en/of ontvangen, onderschept kunnen worden. Afhankelijk van het type behavior moet een bijbehorende interface geïmplementeerd worden in een eigen gedefinieerde class. In het geval van een endpoint-behavior gebruiken we de interface *System.ServiceModel.Description.IEndpointBehavior:* zie codevoorbeeld 8.

De IEndpointBehavior-interface heeft een viertal methodes, waarvan in ons geval alleen *ApplyClientBehavior* een implementatie hoeft te bevatten. Wanneer een endpoint-behavior aan de servicekant toegevoegd moet worden, zouden we *ApplyDispatchBehavior* moeten implementeren. De ApplyClientBehavior-methode biedt toegang tot het *ClientRuntime*-object in de vorm van een inputparameter; zie ook afbeelding 5. De *extensibility-hooks* van het ClientRuntime-object worden beschikbaar gesteld in de vorm van properties. Door een nieuwe instantie van het MyMessageInspector-object aan de collectie van message-inspectors toe te voegen, wordt de extensie geregistreerd. Het endpoint-behavior kunnen we zowel door code als configuratie toevoegen. Vanuit code moeten we een instantie van MyMessageInspectorBehavior toevoegen aan de behaviors-collectie van het endpoint van de proxy.; zie codevoorbeeld 9.

In veel scenario's is het wenselijk om een behavior te kunnen toevoegen zonder code aan te passen. Om dit mogelijk te maken, moet er een zogenaamd custom BehaviorExtenstionElement gemaakt worden. Concreet betekent dit het creëren van een nieuwe class die erft van *System.ServiceModel.Configuration.BehaviorExtensionElement*. De *BehaviorType*-property moet het type van het custom behavior retourneren, *CreateBehavior* een



Afbeelding 5. Client-side model

```
BankingServiceProxy proxy = new BankingServiceProxy();
proxy.Endpoint.Behaviors.Add(new MyMessageInspectorBehavior());

string result = proxy.TransferMoney("A", "B", 100);
```

Codevoorbeeld 9. Endpoint-behavior door middel van code toevoegen

```
class MyMessageInspectorConfig : BehaviorExtensionElement
{
   public override Type BehaviorType
   {
      get { return typeof(MyMessageInspectorBehavior); }
   }

   protected override object CreateBehavior()
   {
      return new MyMessageInspectorBehavior();
   }
}
```
Codevoorbeeld 10. Configuratieondersteuning toevoegen

instantie van hetzelfde type; zie codevoorbeeld 10.
Het custom BehaviorExtensionElement moet allereerst worden opgenomen in het BehaviorExtensions-element in de app.config. Het is tevens mogelijk de extensie voor het gehele systeem beschikbaar te maken door het BehaviorExtensionElement in de machine.config op te nemen. Vervolgens kan het behavior als elk andere behavior in de behaviors-sectie worden gebruikt; zie codevoorbeeld 11.

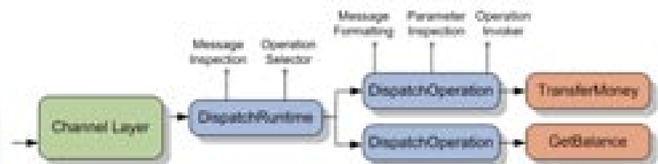
Uitbreiden van de dispatcher (service)

De dispatcher is verantwoordelijk voor de conversie van een bericht, afkomstig van de channel-laag, naar een methodeaanroep van de service-applicatiecode; zie afbeelding 5.
Voor mogelijke responsberichten zal het proces vice versa uitgevoerd worden. Net zoals bij de proxy heeft dit proces een aantal extensiblity-hooks, waar op gelijksoortige wijze gebruik van gemaakt kan worden. Een nieuw extensiepunt ten opzichte van de proxy is de Operation Invoker. De Operation Invoker is verantwoordelijk voor het daadwerkelijk aanroepen van de desbetreffende methode van de service. Deze extensie is te implementeren door middel van de *OperationInvoker*-interface.
In de vorige sectie hebben we gezien hoe een behavior aan de runtime toegevoegd kan worden met behulp van (imperatieve) code en configuratie. Door de behavior werd een extensie aan het ClientRuntime-object toegevoegd. Aan de service-kant hebben we te maken

```
<system.serviceModel>
   <client>
      <endpoint
         address="http://localhost:9000/"
         binding="wsHttpBinding"
         behaviorConfiguration="MyBehaviors"
         contract="Client.BankingService.IBankingService"/>
   </client>

   <!-- Registeren van behaviors-->
      <endpointBehaviors>
         <behavior name="MyBehaviors">
            <MyMessageInspector/>
         </behavior>
      </endpointBehaviors>
   </behaviors>

   <!-- Registreren van custom extensies-->
   <extensions>
      <behaviorExtensions>
         <add name ="MyMessageInspector"
              type ="Client.MyMessageInspectorConfig,
                    Client, Version=1.0.0.0, Culture=neutral,
                    PublicKeyToken=null"/>
      </behaviorExtensions>
   </extensions>
</system.serviceModel>
```
Codevoorbeeld 11. Configureren van het behavior

```
class MyParameterInspector:IParameterInspector
{
   public object BeforeCall(string operationName,
      object[] inputs)
   {
      // Valideer object[] inputs (de parameters) hier !
      return null;
   }

   public void AfterCall(string operationName,
      object[] outputs, object returnValue,
      object correlationState)
      {}
}
```
Codevoorbeeld 12. Custom parameter-inspector

met het DispatchRuntime- en DispatchOperation-object voor individuele operaties. Om een beter beeld te krijgen bij het gebruik van de *extensibility-hooks* op de DispatchOperation implementeren we een simpele parameter-inspector. Deze inspector zal op declaratieve wijze via een custom attribuut zijn toe te passen. Voor het creëren van een custom parameter-inspector gebruiken we de interface *System.ServiceModel.Dispatcher.IParameterInspector*. Deze interface heeft twee methodes die de mogelijkheid bieden om, voor en na het aanroepen van de operatie, de parameters te inspecteren of zelfs aan te passen; zie codevoorbeeld 12.
Om de extensie aan de runtime te kunnen koppelen, moeten we een custom behavior maken. Aangezien deze van toepassing is op een operatie implementeren we de interface *System.ServiceModel.Description.IOperationBehavior.* Deze interface bevat vier methodes, waarvan *ApplyDispatchBehavior* in dit geval de enige is die geïmplementeerd moet worden. Op de inputparameter *dispatchOperation* moet een instantie van de custom parameter-inspector aan de ParameterInspectors-collectie worden toegevoegd. Om de behavior op declaratieve wijze als custom attribuut te kunnen gebruiken, moet er tevens van *System.Attribute* geërfd worden; zie codevoorbeeld 13.



Afbeelding 6. Service-side model

```
class MyParameterInspectorAttribute:Attribute,IOperationBehavior
{
   public void ApplyDispatchBehavior(OperationDescription
      oprationDescription, DispatchOperation dispatchOperation)
   {
      dispatchOperation.ParameterInspectors.Add(
         new InputParameterValidator());
   }

   public void AddBindingParameters(OperationDescription
      operationDescription, BindingParameterCollection bindingParameters)
   {}

   public void ApplyClientBehavior(OperationDescription
      operationDescription, ClientOperation clientOperation)
   {}

   public void Validate(OperationDescription  operationDescription)
   {}
}
```
Codevoorbeeld 13. Custom operation behavior

```
[ServiceContract]
public interface IBankingService
{
   [OperationContract]
   string TransferMoney(
      string creditAccount,
      string debitAccount,
      double amount);
   }

public class BankingService : IBankingService
{
   [MyParameterInspector] // << Voegt de behavior toe
   public string TransferMoney(
      string creditAccount,
      string debitAccount,
      double amount)
   {

      string result = string.Format("{0} dollars has been
         transferred from account {1} to {2}.",
         amount, debitAccount, creditAccount);

         Console.WriteLine(result);
      return result;
   }
}
```

Codevoorbeeld 14. Toepassen custom operation behavior

Doordat het custom operation behavior-type tevens erft van *System. Attribuut* kunnen we het nu toepassen als custom attribuut, zoals te zien is in codevoorbeeld 14. Tijdens het initiële codereflectieproces wordt het custom attribuut gedetecteerd en toegevoegd aan de run-time.

WCF is op zijn eigen extensiepunten gebouwd

In dit artikel is een aantal essentiële aspecten van het service-model besproken. We hebben gezien hoe je op relatief eenvoudige wijze eigen extensies kunt bouwen en hoe je deze door middel van custom behaviors kunt toevoegen aan de runtime. De WCF-architectuur is gebouwd op zijn eigen extensiepunten. Hierdoor zijn nagenoeg alle elementen te vervangen door een eigen implementatie. De channel-layer is zeker net zo uitbreidbaar als het servicemodel en zal daarom dus ook in een apart artikel behandeld worden.

Gijs de Jong is principal consultant bij Microsoft Services. Zijn e-mailadres is gijsdj@microsoft.com.

Referenties

WCF Community Site: http://wcf.netfx3.com/

MSDN Web Services Homepage: http://msdn.microsoft.com/webservices

(advertentie Microsoft Press)

Managing Projects with Microsoft Visual Studio Team System
ISBN: 9780735622166
Author:
Joel Semeniuk; Martin Danner
Pagina's: 272