Stefan Onderstal en Alex Thissen

Ben jij mijn type?

DE EXPLICIETE KEUZE VAN DATATYPES BIJ HET WERKEN MET WEBSERVICES

Sinds de introductie van het .NET Framework zijn steeds meer ontwikkelaars zich op webservices gaan richten. Met het gebruik van webservices is de behoefte ontstaan meer invloed te kunnen uitoefenen op de functionaliteit achter Add Web Reference. De functionaliteit bestaat uit het genereren van een webproxy en classes op basis van de WSDL-definitie. De gegenereerde classes passen echter niet altijd in de architectuur van de applicatie zoals de ontwikkelaar die voor ogen heeft.

In dit artikel beschrijven we welke mogelijkheden er zoal zijn bij het werken met .NET-types, naar aanleiding van een project waar we webservices hebben gebruikt. We beschrijven de mogelijkheden die er zijn, welke keuzes we gemaakt hebben en wat dit heeft opgeleverd. Verder laten we zien hoe deze opties eenvoudiger te realiseren zijn met de introductie van Schema Importer Extensions in .NET 2.0.

Qurius ontwikkelt voor KPN een zogenaamde 'customer self-care portal', waarmee de medewerkers van grootzakelijke klanten (top 500) van KPN via internet bestellingen kunnen plaatsen. Deze webapplicatie moet aansluiten op webservices die ook gebruikt worden door andere applicaties, zoals een business-to-business-applicatie waarbij Microsoft BizTalk Server 2006 wordt ingezet. De customer self-care portal gebruikt webservices om de functionaliteit van de back-end-applicatie te ontsluiten. De webservices zijn ontwikkeld in .NET door een ander team. Dat team heeft er voor gekozen om pure datacontainers te gebruiken die geen publieke methodes of andere logica bevatten. De portal-applicatie werkt met business-entiteiten om de data vast te houden. Deze entiteiten bevatten zowel data als gedrag. De standaardmanier om een webservice te importeren, via wsdl.exe of Add Web Reference, levert nieuwe datatypes op. Deze datatypes houden weliswaar dezelfde data vast, maar voldoen niet aan de eisen om het gedrag en data in business-entiteiten vast te leggen.

Voorwaarden

Het project kende de volgende architecturale voorwaarden en ontwerpkeuzes die belangrijk waren bij het bepalen van de oplossingsrichting:

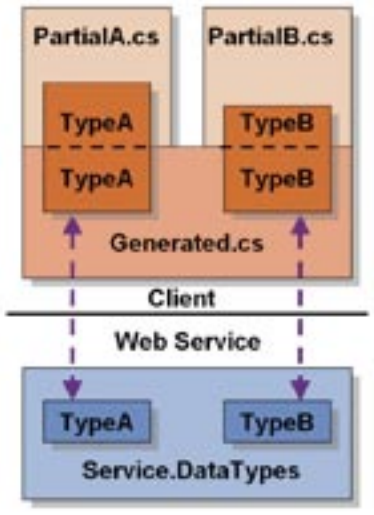
- Er moet een losse koppeling zijn tussen de applicatie en de gebruikte webservices.
- De webapplicatie moet gebruikmaken van entiteiten die zowel data als gedrag bevatten en die in een geïsoleerde assembly gecompileerd en geversioneerd worden.
- De gekozen oplossing moet consistent toegepast kunnen worden op de huidige en toekomstige webservices.
- De hoeveelheid werk voor het realiseren van de optie moet zo laag mogelijk zijn en bovendien goed onderhoudbaar.

We stonden voor de uitdaging om aan de hand van deze voorwaarden een manier en werkwijze te vinden, zodat we met eigen gekozen types (onze business-entiteiten) kunnen werken. Natuurlijk wilden we daarbij gebruikmaken van de eenvoud van Add Web Reference vanuit Visual Studio. We gaan nu kort in op de mogelijkheden die we bekeken hebben voor het introduceren van eigen entiteiten. Al deze opties hebben voor- en nadelen, die we kort benoemen. We hebben drie opties onderscheiden om met types voor een webservice te werken:

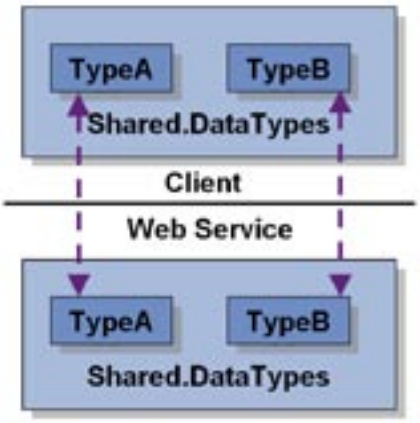
1. Types uit de gegenereerde proxy-assembly met extra logica
2. Types uit een gedeelde assembly
3. Types uit een eigen assembly

Optie 1: Types uit gegenereerde proxy-assembly met extra logica

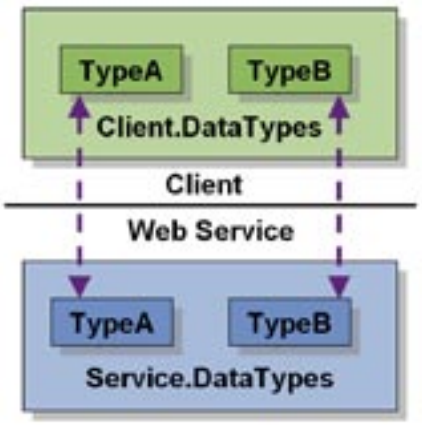
Wanneer de client de types uit de gegenereerde proxycode gebruikt, zijn deze qua datavorm identiek aan de webservicetypes. Ze leven in een eigen namespace en worden mede daarom door de CLR als verschillende types gezien. Deze types bevatten alleen velden en eigenschappen en voldoen niet aan de gestelde eis om ook businesslogica en regels in de types te hebben. De .NET runtime 2.0 kent partial classes, waarbij de code van een class in meer bestanden verspreid kan staan. De gegenereerde classes zijn gemarkeerd met het partial keyword. Dat geeft de mogelijkheid de code in de data-



Afbeelding 1. Optie 1: types uit de gegenereerde proxy-assembly met extra logica



Afbeelding 2. Optie 2: types uit een gedeelde assembly



Afbeelding 3. Optie 3: types uit een eigen assembly

classes uit te breiden met logicacode in extra bestanden. Het scheiden van de gegenereerde en de extra logicacode zorgt er ook voor dat eventuele hergeneratie van code geen nadelige effecten heeft op de logicacode; zie afbeelding 1. Een voordeel van deze optie is dat wijzigingen in het datacontract van de service al tijdens het compileren tot fouten leiden in de client-code ('break early'). Een ander voordeel is dat deze optie relatief eenvoudig is te implementeren. Een groot nadeel is dat de proxycode en de business-entiteiten die zo ontstaan in dezelfde assembly zitten. Dit betekent dat de business-entiteiten en de service-agents fysiek niet van elkaar te scheiden zijn. De enige manier om deze over aparte tiers van de applicatie te verdelen, is het kopiëren van de assembly. Business-entiteiten worden in alle lagen van de applicatie gebruikt. De service-agents zijn dan zelfs in de presentatie-tier terug te vinden. Dat is zeker niet de bedoeling.

Optie 2: Types uit een gedeelde assembly

De client-applicatie kan beschikken over de assembly met de datatypes die de webservice gebruikt. De client en de webservice kunnen dan de datatypes delen. De types uit de proxy-assembly moeten worden omgezet naar de types in de assembly met datatypes. De webservice-proxy moet gebruikmaken van de gedeelde en niet van de gegenereerde datatypes. Het is zelfs overbodig om nog datatypes te laten genereren; zie afbeelding 2. Een voordeel van deze aanpak is dat er geen extra nieuwe code ontwikkeld hoeft te worden. De assembly met datatypes kan direct in de webapplicatie gebruikt worden. Deze optie heeft wel als nadeel dat er een sterke koppeling is met de implementatie in de webservice. Sterker nog, deze is exact hetzelfde. Een van de vier principes uit serviceoriëntatie is: 'Services share schema and contract, not class'. Onze architectuur had niet het streven om servicegeoriënteerd te zijn. Dit ene principe geeft duidelijk aan dat het delen van types niet de manier is om een losse koppeling tussen webservice en client te maken. Toch kan typesharing een goede optie zijn: wanneer de webservices alleen dienen als een remoting implementatie in plaats van bijvoorbeeld .NET Remoting of Enterprise Services. Een nadeel is dat de standaardmanier om met webservices te werken niet past bij deze optie. We willen eigen types kunnen kiezen in plaats van de types die de tools van Visual Studio en de Framework SDK genereren.

Optie 3: Types uit eigen assembly

De derde optie voorziet in een ontkoppeling van de webservice en de client door het definiëren van eigen business- of data-entiteiten. De client is daarmee volledig onafhankelijk van de implementatie van de datatypes aan de zijde van de webservice. Net als bij optie 2 is er bij deze optie geen behoefte aan de gegenereerde types; zie afbeelding 3. Een bijkomend voordeel van deze optie is dat de eigen datatypes in een eigen assembly geplaatst kunnen worden. Hierin kunnen ook nieuwe datatypes geplaatst worden die niet door de webservices gebruikt worden. Net als bij optie 2 is deze optie bij normaal gebruik van Add Web Reference standaard niet mogelijk. Maar, zoals we zo dadelijk zullen zien, zijn er wel degelijk mogelijkheden om je eigen types te kiezen.

Keuzes, keuzes

Wanneer je naar de drie opties kijkt en rekening houdt met de vier eerder genoemde voorwaarden, dan is snel duidelijk dat optie 3 de beste kandidaat is.Het aspect van 'weinig werk en goed onderhoudbaar' verdient nog enige toelichting. Zowel optie 2 als 3 beschrijven ideale situaties die met de standaardfunctionaliteit van wsdl.exe of Add Web Reference niet rechtstreeks mogelijk zijn. De tools genereren standaard zowel de proxy als de datatypes. De proxy werkt dan altijd met de gegenereerde types. In het .NET Framework 1.1 waren er nog geen mogelijkheden invloed uit te oefenen op de codegeneratie van deze tools. Het was daarom nodig een eigen mapping van de gegenereerde types naar de gewenste entiteiten te maken, zoals in afbeelding 4 is te zien.

Het schrijven van de transformatielaag is niet bepaald eenvoudig. Ze is ook lastig te onderhouden bij wijzigingen in het datacontract. Als het mogelijk is, willen wij het genereren en transformeren van de datatypes vermijden. Dat betekent namelijk dat er ongewenste types bestaan en gebruikt worden en dat we runtime de objecten moeten omzetten naar het juiste type. Het is daarom beter dat de datatypes van de webservices direct mappen op onze eigen business-entiteiten, zoals in de figuur van optie 3 aangegeven is. De proxy werkt dan met business-entiteiten. Dit komt onder andere terug in de signatures van de methodes in de proxy. Het uitoefenen van invloed op de keuze van CLR-types tijdens codegeneratie is mogelijk in het .NET Framework 2.0 met behulp van Schema Importer Extensions.

	Optie 1: types uit gegenereerde code met extra logica	Optie 2: types uit gedeelde assembly	Optie 3: types uit eigen assembly
Ontkoppeling client en service	Ja	Nee	Ja
Aparte assembly voor business-entiteiten	Nee	Nee	Ja
Consistentie nu en in de toekomst	Ja	Nee	Ja
Weinig werk en goed onderhoudbaar	Ja	Ja/Nee	Ja/Nee

Tabel 1. Overzicht keuzes

Schema Importer Extensions

Een Schema Importer Extension is een mechanisme waarmee je design-time invloed kunt uitoefenen op het genereren van de proxycode. De XML-serialisatie-API gebruikt geregistreerde Schema Importer Extensions wanneer hij code genereert voor de simpele en complexe types in een XSD-schema. De tools wsdl.exe en Add Web Reference maken gebruik van dit mechanisme. Het resultaat is dat een Schema Importer Extension de mogelijkheid krijgt om tijdens het importeren van het datacontract van de webservice, de types uit het XSD-schema te inspecteren. Het kan dan bestaande datatypes selecteren of nieuwe types creëren via de CodeDOM-provider. De gebruikelijke datatypes worden in beide gevallen niet meer gegenereerd. De proxycode maakt daarna gebruik van de geselecteerde datatypes. De Schema Importer Extension bepaalt alleen welke datatypes de proxy gaat gebruiken, maar kan deze datatypes eventueel ook genereren. Om gebruik te kunnen maken van dit principe, zal een eigen Schema Importer Extension ontwikkeld moeten worden. Wij wilden een generieke implementatie, waarbij we eenvoudig een configureerbare mapping tussen XML-namespaces en CLR-types konden maken. We vonden deze implementatie in een artikel van Jelle Druyts. (Zie de lijst van referenties onderaan dit artikel; red.) Het artikel legt bovendien in een aantal stappen goed uit hoe een Schema Importer Extension functioneert en ingezet kan worden. De SchemaImporterExtension-class is een class in de System.Xml.Serialization.Advanced-namespace. Een Schema Importer Extension wordt gemaakt door te overerven van deze abstracte class. Zoals afbeelding 5 laat zien, zijn er vier methodes op de class gedefinieerd. De ImportSchemaType-methode is de belangrijkste methode om aan te passen. Hierin worden de simpele en complexe types uit het XSD-schema aangeboden. Met behulp van een override kan een ander CLR-type worden benoemd om als container te gebruiken voor de (de)serialisatie van de XML; zie codevoorbeeld 1.

De assembly met de afgeleide SchemaImporterExtension-class moet geregistreerd worden in de Global Assembly Cache (GAC), of moet in de PrivateAssemblies-folder van Visual Studio 2005 geplaatst worden. In het eerste geval zal ook wsdl.exe de Schema Importer Extension kunnen gebruiken. De verwijzing naar deze Schema Importer Extension-assembly neem je op in de runtime-configuratie van de te gebruiken tool; zie codevoorbeeld 2. Het configuratiebestand is devenv.exe.config voor Visual Studio 2005, maar mag ook de machine.config zijn wanneer de Schema Importer Extension vanuit Framework SDK tools ingezet moet worden. Het configuratiebestand zal daarnaast ook de mapping van de verschillende datatypes bevatten; zie codevoorbeeld 3. Het codevoorbeeld toont de mapping van de XML-namespace en het XSD-type naar de Fully Qualified Name (FQN) van het eigen CLR-type. Alle bestaande business-entiteiten in de client-applicatie mappen op deze manier naar de datatypes uit het datacontract van de webservice. Het is belangrijk dat deze mapping secuur gebeurt. Iedere mismatch tussen de XML, de namespaces en het eigen type leidt tot fouten bij (de)serialisatie. Dergelijke fouten manifesteren zich pas tijdens runtime, wanneer data tussen de client en de webservice worden uitgewisseld. Wij zien hier een kans voor tooling die het eenvoudiger maakt de mapping te bewerken.

Twee contracten

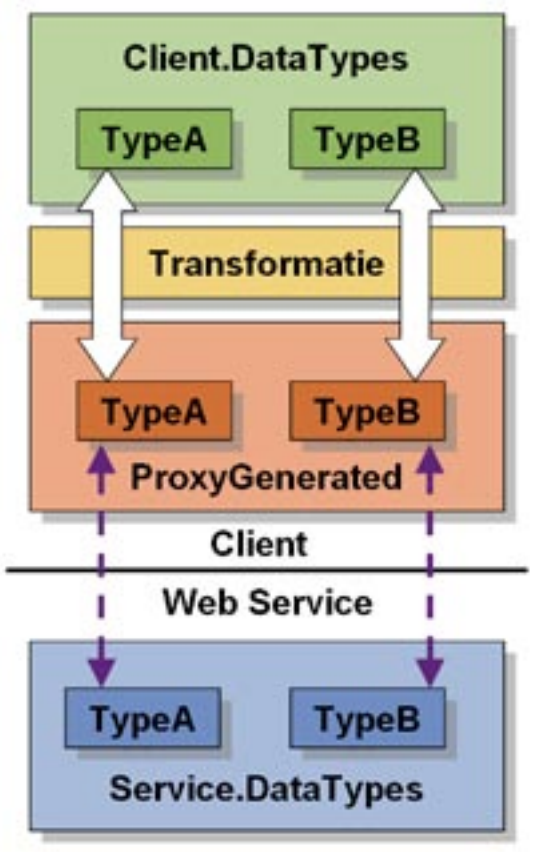
Webservices wisselen SOAP-berichten uit. De webservice gaat daarmee een tweetal contracten aan met de client: een service- en een datacontract. Het servicecontract bepaalt de set van operaties, gevormd door SOAP-berichten, die de webservice aanbiedt. Het datacontract definieert de vorm van de XML voor de informatie die binnen de berichten van een operatie wordt uitgewisseld. De meeste .NET-ontwikkelaars zijn gewend in de custom Common Language Runtime (CLR) met custom-types te werken in plaats van met XML. Het .NET Framework voorziet hierin, omdat het mogelijk is objecten te serialiseren naar XML-formaat en vice versa. Door dit principe toe te passen bij webservices wordt het programmeermodel als vanouds. De definitie van de CLR-types bepaalt impliciet het datacontract. De data die een webservice ontvangt en teruggeeft bestaat uit objecten. De .NET-runtime regelt het parsen van de XML en het bewerken van de SOAP-berichten.

Hetzelfde model is toe te passen bij de client van een webservice. De WSDL-gebaseerde beschrijving van de webservice bevat onder meer het datacontract in de vorm van een XSD-schema. Dit schema kan worden gebruikt om code te genereren die CLR-types beschrijft. De geserialiseerde XML van de objecten komt exact overeen met de XML die de webservice verstuurd en verwacht. Hierdoor kan ook de client-code gebruikmaken van CLR-types en hoeft er niet met XML gewerkt te worden. De tool wsdl.exe genereert de code voor zowel een web-service-proxy op basis van het servicecontract als CLR-types die het datacontract vertegenwoordigen. Visual Studio 2005 kent de optie Add Web Reference om via een dialoog eenvoudig dezelfde functionaliteit aan te roepen. De gegenereerde proxycode en datatypes bevinden zich in de map Web References van het Visual Studio 2005-project. Hiervoor moet de optie 'Show All Files' in de Solution Explorer aanstaan. De code staat in Reference.cs dat in de Reference.map van iedere Web Reference is te vinden.

De generatie van datatypes bij het aanmaken van een webreferentie kan problemen introduceren. De client-applicaties definiëren vaak al datatypes, business-entiteiten of andere vormen van datacontainers. De door wsdl.exe gegenereerde datatypes voldoen meestal niet, omdat ze alleen velden en properties hebben en geen extra functionaliteit en logica bevatten.

Schema Importer Extension geeft invloed

Voor het bouwen van de portal-applicatie voor KPN waren er specifieke eisen ten aanzien van de datatypes en het werken met



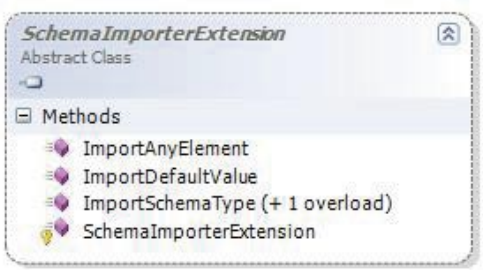
Afbeelding 4. Eigen mapping van de gegenereerde types

```
public override string ImportSchemaType(string name, string ns, XmlSche
maObject context, XmlSchemas schemas, XmlSchemaImporter importer, Code
CompileUnit compileUnit, CodeNamespace mainNamespace, CodeGenerationOp
tions options, CodeDomProvider codeProvider)
{
  try
  {
    foreach (SharedTypeMappingElement mapping in
      SchemaImporterExtensionsConfiguration.Instance.
      SharedTypeMappings)
    {
      // Check if the namespace and type name match.
      if (mapping.XmlTypeName == name && mapping.XmlNamespace == ns)
      {
        // Add an assembly reference.
        if (!string.IsNullOrEmpty(mapping.ClrAssemblyPath))
        {
          compileUnit.ReferencedAssemblies.Add(
            mapping.ClrAssemblyPath);
        }

        // Indicate that no XML schema type should be imported but that
        // a well-known shared CLR type will be used.
        XmlSchemaElement schemaElement = context as XmlSchemaElement;
        if (schemaElement != null && schemaElement.IsNillable &&
            schemaElement.ElementSchemaType is XmlSchemaSimpleType)
        {
          return String.Format("System.Nullable<{0}>",
            mapping.ClrClassName);
        }
        return mapping.ClrClassName;
      }
    }
  }
  catch (Exception exc)
  {
    System.Windows.Forms.MessageBox.Show(exc.Message +
      Environment.NewLine + exc.StackTrace,
      "SharedTypeSchemaImporterExtension Exception",
      System.Windows.Forms.MessageBoxButtons.OK,
      System.Windows.Forms.MessageBoxIcon.Error);
  }

  // No match, delegate to the base class.
  string typeName = base.ImportSchemaType(name, ns, context, schemas,
    importer, compileUnit, mainNamespace, options, codeProvider);
  return typeName;
}
```

Codevoorbeeld 1.



Afbeelding 5. De SchemaImporterExtension-class

webservices. Twee van de drie opties die we tegen elkaar hebben afgewogen, waren relatief bewerkelijk, omdat de standaard tooling van Visual Studio en het .NET Framework niet helemaal meewerkte. Wat doe je dan? Je zoekt naar manieren om de tooling naar je hand te zetten. Je wilt invloed hebben op de keuze van de types tijdens het genereren van proxycode en de daarbij horende data-

```
<system.xml.serialization>
  <schemaImporterExtensions>
    <add name="DmsOnlineSchemaImporterExtension"
         type="Kpn.DmsOnline.Utilities.SchemaImporterExtensions.
         SharedTypeSchemaImporterExtension, Kpn.DmsOnline.Utilities" />
  </schemaImporterExtensions>
</system.xml.serialization>
```

Codevoorbeeld 2.

```
<schemaImporterExtensions>
  <sharedTypeMappings>
    <mapping xmlNamespace=http://schemas.kpn.com/Dms/v1.0.0.0/
Base
             xmlTypeName="BaseEntity"
            clrClassName="Kpn.DmsOnline.BusinessEntities.BaseEntity" />
    <mapping  xmlNamespace="http://schemas.kpn.com/Dms/v1.0.0.0/Base
Classes"
              xmlTypeName="ResponseHeader"
              clrAssemblyPath="C:\KPN\DmsOnline\Source\VS2005\Kpn.
DmsOnline.
              BusinessEntities\bin\Debug\Kpn.DmsOnline.BusinessEnti
ties.dll" clrClassName="Kpn
              .DmsOnline.BusinessEntities.BaseClasses.ResponseHeader"
/>
  </sharedTypeMappings>
</schemaImporterExtensions>
```

Codevoorbeeld 3.

types. Dat kan door het ontwikkelen en inzetten van een Schema Importer Extension. Dit geeft de mogelijkheid je eigen types te kiezen en te gebruiken bij het consumeren van webservices. Het reduceert daarmee de hoeveelheid tijd en werk die nodig zou zijn om een transformatie tussen de niet-gewenste en gewenste datatypes te maken. Bovendien is de oplossing beter te onderhouden. De Schema Importer Extension wordt door meer tools gebruikt bij het genereren van types op basis van een XSD-schema, zoals xsd.exe. Het gebruik van de Schema Importer Extension geeft je in deze gevallen de kans te zeggen: Kijk, dat is nou mijn type!

Stefan Onderstal is sinds januari 2001 werkzaam als software-engineer bij Qurius Advanced Solutions. Stefan.onderstal@qurius.nl

Alex Thissen is werkzaam bij Class-A als trainer en coach. Alex heeft een eigen blog op www.alexthissen.nl.

Referenties

http://www.microsoft.com/belux/msdn/nl/community/columns/jdruyts/wsproxy.mspx
http://www.alexthissen.nl/weblog/DetailsView.aspx?PostingID=eee9d65e-8455-4322-a69e-8bf53dc79db5
http://msdn2.microsoft.com/en-us/library/system.xml.serialization.advanced.schemaimporterextension.aspx